Digital Design and Computer Architecture

SECOND EDITION

A
B
MEM
RF
4
ALU

David Money Harris & Sarah L. Harris

MK
MORGAN KAUFMANN

In Praise of *Digital Design and Computer Architecture*

Harris and Harris have taken the popular pedagogy from Computer Organization and Design *to the next level of refinement, showing in detail how to build a MIPS microprocessor in both SystemVerilog and VHDL. With the exciting opportunity that students have to run large digital designs on modern FGPAs, the approach the authors take in this book is both informative and enlightening.*

David A. Patterson University of California, Berkeley

Digital Design and Computer Architecture *brings a fresh perspective to an old discipline. Many textbooks tend to resemble overgrown shrubs, but Harris and Harris have managed to prune away the deadwood while preserving the fundamentals and presenting them in a contemporary context. In doing so, they offer a text that will benefit students interested in designing solutions for tomorrow's challenges.*

Jim Frenzel University of Idaho

Harris and Harris have a pleasant and informative writing style. Their treatment of the material is at a good level for introducing students to computer engineering with plenty of helpful diagrams. Combinational circuits, microarchitecture, and memory systems are handled particularly well.

James Pinter-Lucke Claremont McKenna College

Harris and Harris have written a book that is very clear and easy to understand. The exercises are well-designed and the real-world examples are a nice touch. The lengthy and confusing explanations often found in similar textbooks are not seen here. It's obvious that the authors have devoted a great deal of time and effort to create an accessible text. I strongly recommend Digital Design and Computer Architecture.

Peiyi Zhao Chapman University

Harris and Harris have created the first book that successfully combines digital system design with computer architecture. Digital Design and Computer Architecture *is a much-welcomed text that extensively explores digital systems designs and explains the MIPS architecture in fantastic detail. I highly recommend this book.*

James E. Stine, Jr., Oklahoma State University

Digital Design and Computer Architecture *is a brilliant book. Harris and Harris seamlessly tie together all the important elements in microprocessor design—transistors, circuits, logic gates, finite state machines, memories, arithmetic units—and conclude with computer architecture. This text is an excellent guide for understanding how complex systems can be flawlessly designed.*

Jaeha Kim Rambus, Inc.

Digital Design and Computer Architecture *is a very well-written book that will appeal to both young engineers who are learning these subjects for the first time and also to the experienced engineers who want to use this book as a reference. I highly recommend it.*

A. Utku Diril Nvidia Corporation

Digital Design and Computer Architecture

Second Edition

About the Authors

David Money Harris is a professor of engineering at Harvey Mudd College. He received his Ph.D. in electrical engineering from Stanford University and his M.Eng. in electrical engineering and computer science from MIT. Before attending Stanford, he worked at Intel as a logic and circuit designer on the Itanium and Pentium II processors. Since then, he has consulted at Sun Microsystems, Hewlett-Packard, Evans & Sutherland, and other design companies.

David's passions include teaching, building chips, and exploring the outdoors. When he is not at work, he can usually be found hiking, mountaineering, or rock climbing. He particularly enjoys hiking with his three sons. David holds about a dozen patents and is the author of three other textbooks on chip design, as well as four guidebooks to the Southern California mountains.

Sarah L. Harris is an associate professor of engineering at Harvey Mudd College. She received her Ph.D. and M.S. in electrical engineering from Stanford University. Before attending Stanford, she received a B.S. in electrical and computer engineering from Brigham Young University. Sarah has also worked at Hewlett-Packard, the San Diego Supercomputer Center, and Nvidia.

Sarah loves teaching and experimenting in the lab. When she is not working or running after her two sons, you can find her playing music with friends, hiking, kayaking, biking, and traveling.

Digital Design and Computer Architecture

Second Edition

David Money Harris
Sarah L. Harris

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MORGAN KAUFMANN

Acquiring Editor: Todd Green
Development Editor: Nathaniel McFadden
Project Manager: Danielle S. Miller
Designer: Dennis Schaefer

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA

© 2013 Elsevier, Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: *www.elsevier.com/permissions*.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Certain materials contained herein are reprinted with the permission of Microchip Technology Incorporated. No further reprints or reproductions may be made of said materials without Microchip Technology Inc.'s prior written consent.

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data
Application submitted

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

ISBN: 978-0-12-394424-5

For information on all MK publications visit
our website at *www.mkp.com*

Printed in the United States of America
12 13 14 15 10 9 8 7 6 5 4 3 2 1

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER BOOK AID International Sabre Foundation

To my family, Jennifer, Abraham, Samuel, and Benjamin
– DMH

To Ivan and Ocaan, who defy logic
– SLH

This page intentionally left blank

Contents

Preface ... xix

Features ... xx
Online Supplements ... xxi
How to Use the Software Tools in a Course ... xxii
Labs ... xxiii
Bugs ... xxiii
Acknowledgments ... xxiii

Chapter 1 From Zero to One ... 3

1.1 The Game Plan ... 3
1.2 The Art of Managing Complexity ... 4
  1.2.1 *Abstraction* ... 4
  1.2.2 *Discipline* ... 5
  1.2.3 *The Three-Y's* ... 6
1.3 The Digital Abstraction ... 7
1.4 Number Systems ... 9
  1.4.1 *Decimal Numbers* ... 9
  1.4.2 *Binary Numbers* ... 9
  1.4.3 *Hexadecimal Numbers* ... 11
  1.4.4 *Bytes, Nibbles, and All That Jazz* ... 13
  1.4.5 *Binary Addition* ... 14
  1.4.6 *Signed Binary Numbers* ... 15
1.5 Logic Gates ... 19
  1.5.1 *NOT Gate* ... 20
  1.5.2 *Buffer* ... 20
  1.5.3 *AND Gate* ... 20
  1.5.4 *OR Gate* ... 21
  1.5.5 *Other Two-Input Gates* ... 21
  1.5.6 *Multiple-Input Gates* ... 21
1.6 Beneath the Digital Abstraction ... 22
  1.6.1 *Supply Voltage* ... 22
  1.6.2 *Logic Levels* ... 22
  1.6.3 *Noise Margins* ... 23
  1.6.4 *DC Transfer Characteristics* ... 24
  1.6.5 *The Static Discipline* ... 24

1.7 CMOS Transistors 26
1.7.1 *Semiconductors* 27
1.7.2 *Diodes* 27
1.7.3 *Capacitors* 28
1.7.4 *nMOS and pMOS Transistors* 28
1.7.5 *CMOS NOT Gate* 31
1.7.6 *Other CMOS Logic Gates* 31
1.7.7 *Transmission Gates* 33
1.7.8 *Pseudo-nMOS Logic* 33
1.8 Power Consumption 34
1.9 Summary and a Look Ahead 35
Exercises 37
Interview Questions 52

Chapter 2 Combinational Logic Design 55

2.1 Introduction 55
2.2 Boolean Equations 58
2.2.1 *Terminology* 58
2.2.2 *Sum-of-Products Form* 58
2.2.3 *Product-of-Sums Form* 60
2.3 Boolean Algebra 60
2.3.1 *Axioms* 61
2.3.2 *Theorems of One Variable* 61
2.3.3 *Theorems of Several Variables* 62
2.3.4 *The Truth Behind It All* 64
2.3.5 *Simplifying Equations* 65
2.4 From Logic to Gates 66
2.5 Multilevel Combinational Logic 69
2.5.1 *Hardware Reduction* 70
2.5.2 *Bubble Pushing* 71
2.6 X's and Z's, Oh My 73
2.6.1 *Illegal Value: X* 73
2.6.2 *Floating Value: Z* 74
2.7 Karnaugh Maps 75
2.7.1 *Circular Thinking* 76
2.7.2 *Logic Minimization with K-Maps* 77
2.7.3 *Don't Cares* 81
2.7.4 *The Big Picture* 82
2.8 Combinational Building Blocks 83
2.8.1 *Multiplexers* 83
2.8.2 *Decoders* 86
2.9 Timing 88
2.9.1 *Propagation and Contamination Delay* 88
2.9.2 *Glitches* 92

2.10 Summary ... 95
Exercises ... 97
Interview Questions ... 106

Chapter 3 Sequential Logic Design ... 109

3.1 Introduction ... 109
3.2 Latches and Flip-Flops ... 109
    3.2.1 *SR Latch* ... 111
    3.2.2 *D Latch* ... 113
    3.2.3 *D FIip-Flop* ... 114
    3.2.4 *Register* ... 114
    3.2.5 *Enabled Flip-Flop* ... 115
    3.2.6 *Resettable Flip-Flop* ... 116
    3.2.7 *Transistor-Level Latch and Flip-Flop Designs* ... 116
    3.2.8 *Putting It All Together* ... 118
3.3 Synchronous Logic Design ... 119
    3.3.1 *Some Problematic Circuits* ... 119
    3.3.2 *Synchronous Sequential Circuits* ... 120
    3.3.3 *Synchronous and Asynchronous Circuits* ... 122
3.4 Finite State Machines ... 123
    3.4.1 *FSM Design Example* ... 123
    3.4.2 *State Encodings* ... 129
    3.4.3 *Moore and Mealy Machines* ... 132
    3.4.4 *Factoring State Machines* ... 134
    3.4.5 *Deriving an FSM from a Schematic* ... 137
    3.4.6 *FSM Review* ... 140
3.5 Timing of Sequential Logic ... 141
    3.5.1 *The Dynamic Discipline* ... 142
    3.5.2 *System Timing* ... 142
    3.5.3 *Clock Skew* ... 148
    3.5.4 *Metastability* ... 151
    3.5.5 *Synchronizers* ... 152
    3.5.6 *Derivation of Resolution Time* ... 154
3.6 Parallelism ... 157
3.7 Summary ... 161
Exercises ... 162
Interview Questions ... 171

Chapter 4 Hardware Description Languages ... 173

4.1 Introduction ... 173
    4.1.1 *Modules* ... 173
    4.1.2 *Language Origins* ... 174
    4.1.3 *Simulation and Synthesis* ... 175

4.2 Combinational Logic 177
    4.2.1 *Bitwise Operators* 177
    4.2.2 *Comments and White Space* 180
    4.2.3 *Reduction Operators* 180
    4.2.4 *Conditional Assignment* 181
    4.2.5 *Internal Variables* 182
    4.2.6 *Precedence* 184
    4.2.7 *Numbers* 185
    4.2.8 *Z's and X's* 186
    4.2.9 *Bit Swizzling* 188
    4.2.10 *Delays* 188
4.3 Structural Modeling 190
4.4 Sequential Logic 193
    4.4.1 *Registers* 193
    4.4.2 *Resettable Registers* 194
    4.4.3 *Enabled Registers* 196
    4.4.4 *Multiple Registers* 197
    4.4.5 *Latches* 198
4.5 More Combinational Logic 198
    4.5.1 *Case Statements* 201
    4.5.2 *If Statements* 202
    4.5.3 *Truth Tables with Don't Cares* 205
    4.5.4 *Blocking and Nonblocking Assignments* 205
4.6 Finite State Machines 209
4.7 Data Types 213
    4.7.1 *SystemVerilog* 214
    4.7.2 *VHDL* 215
4.8 Parameterized Modules 217
4.9 Testbenches 220
4.10 Summary 224
Exercises 226
Interview Questions 237

Chapter 5 Digital Building Blocks 239

5.1 Introduction 239
5.2 Arithmetic Circuits 239
    5.2.1 *Addition* 239
    5.2.2 *Subtraction* 246
    5.2.3 *Comparators* 246
    5.2.4 *ALU* 248
    5.2.5 *Shifters and Rotators* 250
    5.2.6 *Multiplication* 252

- 5.2.7 *Division* 253
- 5.2.8 *Further Reading* 254
5.3 Number Systems 255
- 5.3.1 *Fixed-Point Number Systems* 255
- 5.3.2 *Floating-Point Number Systems* 256
5.4 Sequential Building Blocks 260
- 5.4.1 *Counters* 260
- 5.4.2 *Shift Registers* 261
5.5 Memory Arrays 263
- 5.5.1 *Overview* 263
- 5.5.2 *Dynamic Random Access Memory (DRAM)* 266
- 5.5.3 *Static Random Access Memory (SRAM)* 266
- 5.5.4 *Area and Delay* 266
- 5.5.5 *Register Files* 267
- 5.5.6 *Read Only Memory* 268
- 5.5.7 *Logic Using Memory Arrays* 270
- 5.5.8 *Memory HDL* 270
5.6 Logic Arrays 272
- 5.6.1 *Programmable Logic Array* 272
- 5.6.2 *Field Programmable Gate Array* 274
- 5.6.3 *Array Implementations* 279
5.7 Summary 280
Exercises 282
Interview Questions 293

Chapter 6 Architecture 295

6.1 Introduction 295
6.2 Assembly Language 296
- 6.2.1 *Instructions* 296
- 6.2.2 *Operands: Registers, Memory, and Constants* 298
6.3 Machine Language 305
- 6.3.1 *R-Type Instructions* 305
- 6.3.2 *I-Type Instructions* 307
- 6.3.3 *J-Type Instructions* 308
- 6.3.4 *Interpreting Machine Language Code* 308
- 6.3.5 *The Power of the Stored Program* 309
6.4 Programming 310
- 6.4.1 *Arithmetic/Logical Instructions* 310
- 6.4.2 *Branching* 314
- 6.4.3 *Conditional Statements* 316
- 6.4.4 *Getting Loopy* 317
- 6.4.5 *Arrays* 320
- 6.4.6 *Function Calls* 325

6.5 Addressing Modes 333
6.6 Lights, Camera, Action: Compiling, Assembling, and Loading. . . 336
6.6.1 *The Memory Map* 336
6.6.2 *Translating and Starting a Program* 337
6.7 Odds and Ends 342
6.7.1 *Pseudoinstructions* 342
6.7.2 *Exceptions* 343
6.7.3 *Signed and Unsigned Instructions* 344
6.7.4 *Floating-Point Instructions* 346
6.8 Real-World Perspective: x86 Architecture 347
6.8.1 *x86 Registers* 348
6.8.2 *x86 Operands* 348
6.8.3 *Status Flags* 350
6.8.4 *x86 Instructions* 350
6.8.5 *x86 Instruction Encoding* 352
6.8.6 *Other x86 Peculiarities* 354
6.8.7 *The Big Picture* 354
6.9 Summary 355
Exercises 357
Interview Questions 369

Chapter 7 Microarchitecture 371

7.1 Introduction 371
7.1.1 *Architectural State and Instruction Set* 371
7.1.2 *Design Process* 372
7.1.3 *MIPS Microarchitectures* 374
7.2 Performance Analysis 374
7.3 Single-Cycle Processor 376
7.3.1 *Single-Cycle Datapath* 376
7.3.2 *Single-Cycle Control* 382
7.3.3 *More Instructions* 385
7.3.4 *Performance Analysis* 388
7.4 Multicycle Processor 389
7.4.1 *Multicycle Datapath* 390
7.4.2 *Multicycle Control* 396
7.4.3 *More Instructions* 403
7.4.4 *Performance Analysis* 405
7.5 Pipelined Processor 409
7.5.1 *Pipelined Datapath* 412
7.5.2 *Pipelined Control* 413
7.5.3 *Hazards* 414
7.5.4 *More Instructions* 426
7.5.5 *Performance Analysis* 426

7.6 HDL Representation 429
7.6.1 *Single-Cycle Processor* 430
7.6.2 *Generic Building Blocks* 434
7.6.3 *Testbench* 437
7.7 Exceptions 440
7.8 Advanced Microarchitecture 444
7.8.1 *Deep Pipelines* 444
7.8.2 *Branch Prediction* 446
7.8.3 *Superscalar Processor* 447
7.8.4 *Out-of-Order Processor* 450
7.8.5 *Register Renaming* 452
7.8.6 *Single Instruction Multiple Data* 454
7.8.7 *Multithreading* 455
7.8.8 *Homogeneous Multiprocessors* 456
7.8.9 *Heterogeneous Multiprocessors* 456
7.9 Real-World Perspective: x86 Microarchitecture 458
7.10 Summary 465
Exercises 467
Interview Questions 473

Chapter 8 Memory and I/O Systems 475

8.1 Introduction 475
8.2 Memory System Performance Analysis 479
8.3 Caches 480
8.3.1 *What Data is Held in the Cache?* 481
8.3.2 *How is Data Found?* 482
8.3.3 *What Data is Replaced?* 490
8.3.4 *Advanced Cache Design* 491
8.3.5 *The Evolution of MIPS Caches* 495
8.4 Virtual Memory 496
8.4.1 *Address Translation* 498
8.4.2 *The Page Table* 500
8.4.3 *The Translation Lookaside Buffer* 502
8.4.4 *Memory Protection* 503
8.4.5 *Replacement Policies* 504
8.4.6 *Multilevel Page Tables* 504
8.5 I/O Introduction 506
8.6 Embedded I/O Systems 508
8.6.1 *PIC32MX675F512H Microcontroller* 509
8.6.2 *General-Purpose Digital I/O* 513
8.6.3 *Serial I/O* 515
8.6.4 *Timers* 527

8.6.5 *Interrupts* 529
8.6.6 *Analog I/O* 531
8.6.7 *Other Microcontroller Peripherals* 537
8.7 PC I/O Systems 558
8.7.1 *USB* 559
8.7.2 *PCI and PCI Express* 560
8.7.3 *DDR3 Memory* 561
8.7.4 *Networking* 561
8.7.5 *SATA* 562
8.7.6 *Interfacing to a PC* 562
8.8 Real-World Perspective: x86 Memory and I/O Systems 564
8.8.1 *x86 Cache Systems* 564
8.8.2 *x86 Virtual Memory* 567
8.8.3 *x86 Programmed I/O* 567
8.9 Summary 568
Epilogue 568
Exercises 570
Interview Questions 580

Appendix A Digital System Implementation 583

A.1 Introduction 583
A.2 74xx Logic 583
A.2.1 *Logic Gates* 584
A.2.2 *Other Functions* 584
A.3 Programmable Logic 584
A.3.1 *PROMs* 584
A.3.2 *PLAs* 588
A.3.3 *FPGAs* 589
A.4 Application-Specific Integrated Circuits 591
A.5 Data Sheets 591
A.6 Logic Families 597
A.7 Packaging and Assembly 599
A.8 Transmission Lines 602
A.8.1 *Matched Termination* 604
A.8.2 *Open Termination* 606
A.8.3 *Short Termination* 607
A.8.4 *Mismatched Termination* 607
A.8.5 *When to Use Transmission Line Models* 610
A.8.6 *Proper Transmission Line Terminations* 610
A.8.7 *Derivation of* Z_0 612
A.8.8 *Derivation of the Reflection Coefficient* 613
A.8.9 *Putting It All Together* 614
A.9 Economics 615

Appendix B MIPS Instructions 619

Appendix C C Programming 623

C.1 Introduction 623
C.2 Welcome to C 625
  C.2.1 *C Program Dissection* 625
  C.2.2 *Running a C Program* 626
C.3 Compilation 627
  C.3.1 *Comments* 627
  C.3.2 `#define` 627
  C.3.3 `#include` 628
C.4 Variables 629
  C.4.1 *Primitive Data Types* 630
  C.4.2 *Global and Local Variables* 631
  C.4.3 *Initializing Variables* 633
C.5 Operators 633
C.6 Function Calls 637
C.7 Control-Flow Statements 638
  C.7.1 *Conditional Statements* 639
  C.7.2 *Loops* 641
C.8 More Data Types 643
  C.8.1 *Pointers* 643
  C.8.2 *Arrays* 645
  C.8.3 *Characters* 649
  C.8.4 *Strings* 650
  C.8.5 *Structures* 651
  C.8.6 `typedef` 653
  C.8.7 *Dynamic Memory Allocation* 654
  C.8.8 *Linked Lists* 655
C.9 Standard Libraries 657
  C.9.1 *stdio* 657
  C.9.2 *stdlib* 662
  C.9.3 *math* 664
  C.9.4 *string* 665
C.10 Compiler and Command Line Options 665
  C.10.1 *Compiling Multiple C Source Files* 665
  C.10.2 *Compiler Options* 665
  C.10.3 *Command Line Arguments* 666
C.11 Common Mistakes 667

Further Reading 673

Index 675

This page intentionally left blank

Preface

Why publish yet another book on digital design and computer architecture? There are dozens of good books in print on digital design. There are also several good books about computer architecture, especially the classic texts of Patterson and Hennessy. This book is unique in its treatment in that it presents digital logic design from the perspective of computer architecture, starting at the beginning with 1's and 0's, and leading students through the design of a MIPS microprocessor.

We have used several editions of Patterson and Hennessy's *Computer Organization and Design* (*COD*) for many years at Harvey Mudd College. We particularly like their coverage of the MIPS architecture and microarchitecture because MIPS is a commercially successful microprocessor architecture, yet it is simple enough to clearly explain and build in an introductory class. Because our class has no prerequisites, the first half of the semester is dedicated to digital design, which is not covered by *COD*. Other universities have indicated a need for a book that combines digital design and computer architecture. We have undertaken to prepare such a book.

We believe that building a microprocessor is a special rite of passage for engineering and computer science students. The inner workings of a processor seem almost magical to the uninitiated, yet prove to be straightforward when carefully explained. Digital design in itself is a powerful and exciting subject. Assembly language programming unveils the inner language spoken by the processor. Microarchitecture is the link that brings it all together.

This book is suitable for a rapid-paced, single-semester introduction to digital design and computer architecture or for a two-quarter or two-semester sequence giving more time to digest the material and experiment in the lab. The course can be taught without prerequisites. The material is usually taught at the sophomore- or junior-year level, but may also be accessible to bright freshmen.

FEATURES

This book offers a number of special features.

Side-by-Side Coverage of SystemVerilog and VHDL

Hardware description languages (HDLs) are at the center of modern digital design practices. Unfortunately, designers are evenly split between the two dominant languages, SystemVerilog and VHDL. This book introduces HDLs in Chapter 4 as soon as combinational and sequential logic design has been covered. HDLs are then used in Chapters 5 and 7 to design larger building blocks and entire processors. Nevertheless, Chapter 4 can be skipped and the later chapters are still accessible for courses that choose not to cover HDLs.

This book is unique in its side-by-side presentation of SystemVerilog and VHDL, enabling the reader to learn the two languages. Chapter 4 describes principles applying to both HDLs, then provides language-specific syntax and examples in adjacent columns. This side-by-side treatment makes it easy for an instructor to choose either HDL, and for the reader to transition from one to the other, either in a class or in professional practice.

Classic MIPS Architecture and Microarchitecture

Chapters 6 and 7 focus on the MIPS architecture adapted from the treatment of Patterson and Hennessy. MIPS is an ideal architecture because it is a real architecture shipped in millions of products yearly, yet it is streamlined and easy to learn. Moreover, hundreds of universities around the world have developed pedagogy, labs, and tools around the MIPS architecture.

Real-World Perspectives

Chapters 6, 7, and 8 illustrate the architecture, microarchitecture, and memory hierarchy of Intel x86 processors. Chapter 8 also describes peripherals in the context of Microchip's PIC32 microcontroller. These real-world perspective chapters show how the concepts in the chapters relate to the chips found in many PCs and consumer electronics.

Accessible Overview of Advanced Microarchitecture

Chapter 7 includes an overview of modern high-performance microarchitectural features including branch prediction, superscalar and out-of-order operation, multithreading, and multicore processors. The treatment is accessible to a student in a first course and shows how the microarchitectures in the book can be extended to modern processors.

End-of-Chapter Exercises and Interview Questions

The best way to learn digital design is to do it. Each chapter ends with numerous exercises to practice the material. The exercises are followed by a set of interview questions that our industrial colleagues have asked students applying for work in the field. These questions provide a helpful glimpse into the types of problems job applicants will typically encounter during the interview process. (Exercise solutions are available via the book's companion and instructor webpages. For more details, see the next section, Online Supplements.)

ONLINE SUPPLEMENTS

Supplementary materials are available online at *textbooks.elsevier.com/9780123944245*. This companion site (accessible to all readers) includes:

- Solutions to odd-numbered exercises
- Links to professional-strength computer-aided design (CAD) tools from Altera® and Synopsys®
- Link to QtSpim (referred to generically as SPIM), a MIPS simulator
- Hardware description language (HDL) code for the MIPS processor
- Altera Quartus II helpful hints
- Microchip MPLAB IDE helpful hints
- Lecture slides in PowerPoint (PPT) format
- Sample course and lab materials
- List of errata

The instructor site (linked to the companion site and accessible to adopters who register at *textbooks.elsevier.com*) includes:

- Solutions to all exercises
- Links to professional-strength computer-aided design (CAD) tools from Altera® and Synopsys®. (Synopsys offers Synplify® Premier to qualified universities in a package of 50 licenses. For more information on the Synopsys University program, go to the instructor site for this book.)
- Figures from the text in JPG and PPT formats

Additional details on using the Altera, Synopsys, Microchip, and QtSpim tools in your course are provided in the next section. Details on the sample lab materials are also provided here.

HOW TO USE THE SOFTWARE TOOLS IN A COURSE

Altera Quartus II

Quartus II Web Edition is a free version of the professional-strength Quartus™ II FPGA design tools. It allows students to enter their digital designs in schematic or using either the SystemVerilog or VHDL hardware description language (HDL). After entering the design, students can simulate their circuits using ModelSim™-Altera Starter Edition, which is available with the Altera Quartus II Web Edition. Quartus II Web Edition also includes a built-in logic synthesis tool supporting both SystemVerilog and VHDL.

The difference between Web Edition and Subscription Edition is that Web Edition supports a subset of the most common Altera FPGAs. The difference between ModelSim-Altera Starter Edition and ModelSim commercial versions is that Starter Edition degrades performance for simulations with more than 10,000 lines of HDL.

Microchip MPLAB IDE

Microchip MPLAB Integrated Development Environment (IDE) is a tool for programming PIC microcontrollers and is available for free download. MPLAB integrates program writing, compiling, simulating, and debugging into a single interface. It includes a C compiler and debugger, allowing the students to develop C and assembly programs, compile them, and optionally program them onto a PIC microcontroller.

Optional Tools: Synplify Premier and QtSpim

Synplify Premier and QtSpim are optional tools that can be used with this material.

The Synplify Premier product is a synthesis and debug environment for FPGA and CPLD design. Included is HDL Analyst, a unique graphical HDL analysis tool that automatically generates schematic views of the design with cross-probing back to the HDL source code. This is immensely useful in the learning and debugging process.

Synopsys offers Synplify Premier to qualified universities in a package of 50 licenses. For more information on the Synopsys University program or the Synopsys FPGA design software, visit the instructor site for this book (*textbooks.elsevier.com/9780123944245*).

QtSpim, also called simply SPIM, is a MIPS simulator that runs MIPS assembly code. Students enter their MIPS assembly code into a text file and run it using QtSpim. QtSpim displays the instructions, memory, and register values. Links to the user's manual and example files are available at the companion site (*textbooks.elsevier.com/9780123944245*).

LABS

The companion site includes links to a series of labs that cover topics from digital design through computer architecture. The labs teach students how to use the Quartus II tools to enter, simulate, synthesize, and implement their designs. The labs also include topics on C and assembly language programming using the Microchip MPLAB IDE.

After synthesis, students can implement their designs using the Altera DE2 Development and Education Board. This powerful and competitively priced board is available from *www.altera.com*. The board contains an FPGA that can be programmed to implement student designs. We provide labs that describe how to implement a selection of designs on the DE2 Board using Cyclone II Web Edition.

To run the labs, students will need to download and install Altera Quartus II Web Edition and Microchip MPLAB IDE. Instructors may also choose to install the tools on lab machines. The labs include instructions on how to implement the projects on the DE2 Board. The implementation step may be skipped, but we have found it of great value.

We have tested the labs on Windows, but the tools are also available for Linux.

BUGS

As all experienced programmers know, any program of significant complexity undoubtedly contains bugs. So too do books. We have taken great care to find and squash the bugs in this book. However, some errors undoubtedly do remain. We will maintain a list of errata on the book's webpage.

Please send your bug reports to *ddcabugs@onehotlogic.com*. The first person to report a substantive bug with a fix that we use in a future printing will be rewarded with a $1 bounty!

ACKNOWLEDGMENTS

First and foremost, we thank David Patterson and John Hennessy for their pioneering MIPS microarchitectures described in their *Computer Organization and Design* textbook. We have taught from various editions of their book for many years. We appreciate their gracious support of this book and their permission to build on their microarchitectures.

Duane Bibby, our favorite cartoonist, labored long and hard to illustrate the fun and adventure of digital design. We also appreciate the enthusiasm of Nate McFadden, Todd Green, Danielle Miller, Robyn Day, and the rest of the team at Morgan Kaufmann who made this book happen.

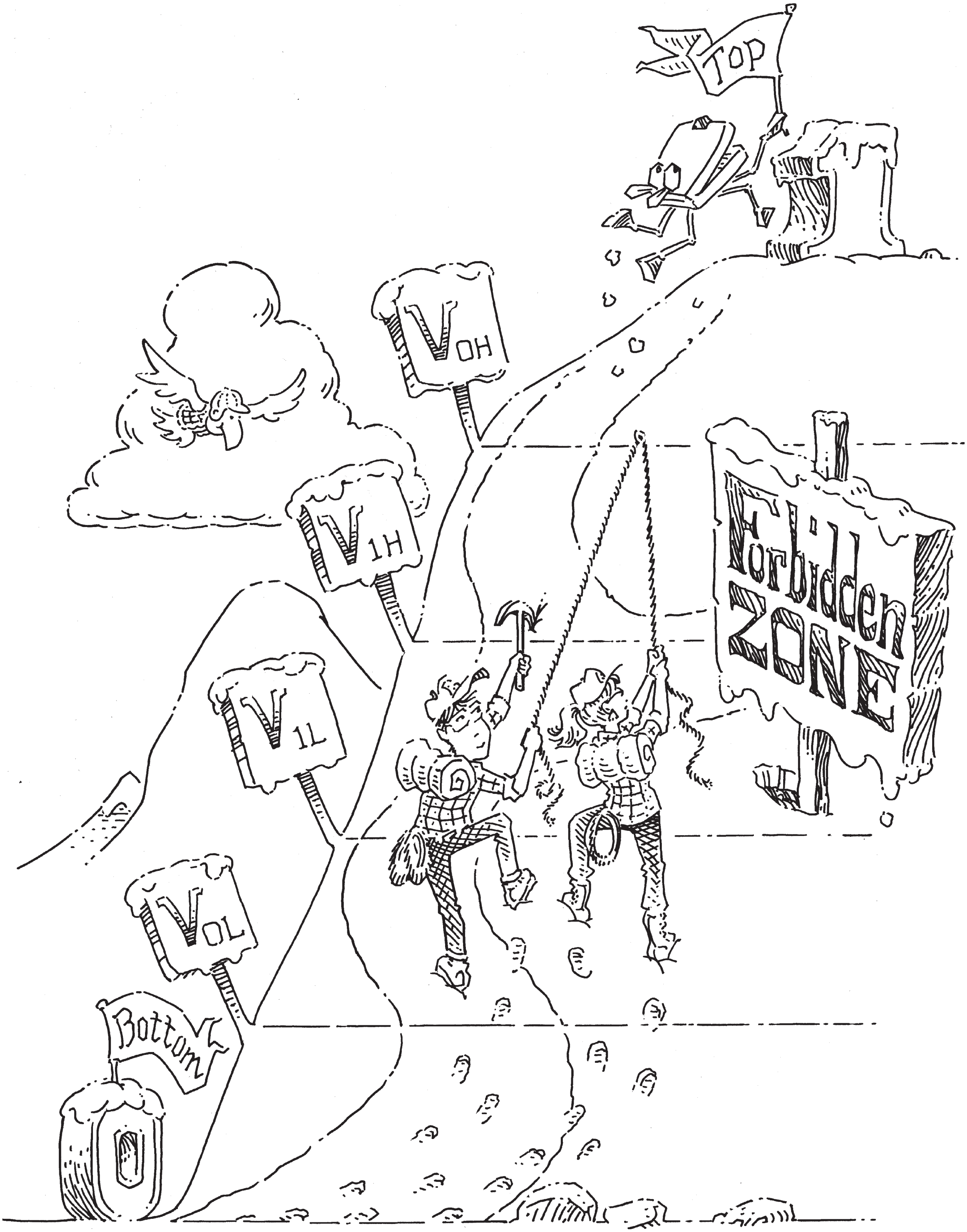
We'd like to thank Matthew Watkins who contributed the section on Heterogeneous Multiprocessors in Chapter 7. We also appreciate the work of Chris Parks, Carl Pearson, and Johnathan Chai who tested code and developed content for the second edition.

Numerous reviewers substantially improved the book. They include John Barr, Jack V. Briner, Andrew C. Brown, Carl Baumgaertner, A. Utku Diril, Jim Frenzel, Jaeha Kim, Phillip King, James Pinter-Lucke, Amir Roth, Z. Jerry Shi, James E. Stine, Luke Teyssier, Peiyi Zhao, Zach Dodds, Nathaniel Guy, Aswin Krishna, Volnei Pedroni, Karl Wang, Ricardo Jasinski, and an anonymous reviewer.

We also appreciate the students in our course at Harvey Mudd College who have given us helpful feedback on drafts of this textbook. Of special note are Matt Weiner, Carl Walsh, Andrew Carter, Casey Schilling, Alice Clifton, Chris Acon, and Stephen Brawner.

And, last but not least, we both thank our families for their love and support.

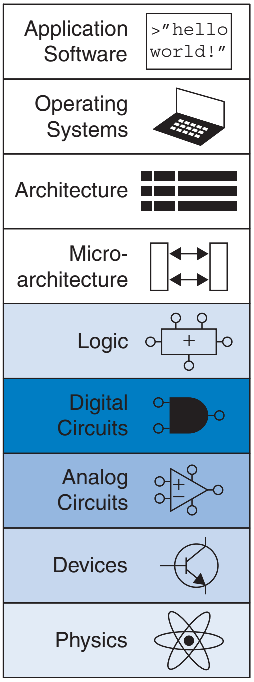
This page intentionally left blank


Top
V_OH
V_1H
Forbidden ZONE
V_1L
V_OL
Bottom
0

1

From Zero to One

1.1 The Game Plan
1.2 The Art of Managing Complexity
1.3 The Digital Abstraction
1.4 Number Systems
1.5 Logic Gates
1.6 Beneath the Digital Abstraction
1.7 CMOS Transistors*
1.8 Power Consumption*
1.9 Summary and a Look Ahead
Exercises
Interview Questions



1.1 THE GAME PLAN

Microprocessors have revolutionized our world during the past three decades. A laptop computer today has far more capability than a room-sized mainframe of yesteryear. A luxury automobile contains about 50 microprocessors. Advances in microprocessors have made cell phones and the Internet possible, have vastly improved medicine, and have transformed how war is waged. Worldwide semiconductor industry sales have grown from US $21 billion in 1985 to $300 billion in 2011, and microprocessors are a major segment of these sales. We believe that microprocessors are not only technically, economically, and socially important, but are also an intrinsically fascinating human invention. By the time you finish reading this book, you will know how to design and build your own microprocessor. The skills you learn along the way will prepare you to design many other digital systems.

We assume that you have a basic familiarity with electricity, some prior programming experience, and a genuine interest in understanding what goes on under the hood of a computer. This book focuses on the design of digital systems, which operate on 1's and 0's. We begin with digital logic gates that accept 1's and 0's as inputs and produce 1's and 0's as outputs. We then explore how to combine logic gates into more complicated modules such as adders and memories. Then we shift gears to programming in assembly language, the native tongue of the microprocessor. Finally, we put gates together to build a microprocessor that runs these assembly language programs.

A great advantage of digital systems is that the building blocks are quite simple: just 1's and 0's. They do not require grungy mathematics or a profound knowledge of physics. Instead, the designer's challenge is to combine these simple blocks into complicated systems. A microprocessor may be the first system that you build that is too complex to fit in

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00001-X
© 2013 Elsevier, Inc. All rights reserved.

your head all at once. One of the major themes weaved through this book is how to manage complexity.

1.2 THE ART OF MANAGING COMPLEXITY

One of the characteristics that separates an engineer or computer scientist from a layperson is a systematic approach to managing complexity. Modern digital systems are built from millions or billions of transistors. No human being could understand these systems by writing equations describing the movement of electrons in each transistor and solving all of the equations simultaneously. You will need to learn to manage complexity to understand how to build a microprocessor without getting mired in a morass of detail.

1.2.1 Abstraction

The critical technique for managing complexity is *abstraction:* hiding details when they are not important. A system can be viewed from many different levels of abstraction. For example, American politicians abstract the world into cities, counties, states, and countries. A county contains multiple cities and a state contains many counties. When a politician is running for president, the politician is mostly interested in how the state as a whole will vote, rather than how each county votes, so the state is the most useful level of abstraction. On the other hand, the Census Bureau measures the population of every city, so the agency must consider the details of a lower level of abstraction.

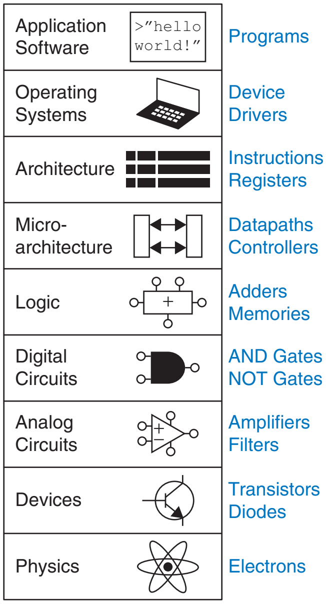

Application Software
>"hello world!"
Programs
Operating Systems
Device Drivers
Architecture
Instructions Registers
Micro-architecture
Datapaths Controllers
Logic
+
Adders Memories
Digital Circuits
AND Gates NOT Gates
Analog Circuits
+
−
Amplifiers Filters
Devices
Transistors Diodes
Physics
Electrons

Figure 1.1 **Levels of abstraction for an electronic computing system**

Figure 1.1 illustrates levels of abstraction for an electronic computer system along with typical building blocks at each level. At the lowest level of abstraction is the physics, the motion of electrons. The behavior of electrons is described by quantum mechanics and Maxwell's equations. Our system is constructed from electronic *devices* such as transistors (or vacuum tubes, once upon a time). These devices have well-defined connection points called *terminals* and can be modeled by the relationship between voltage and current as measured at each terminal. By abstracting to this device level, we can ignore the individual electrons. The next level of abstraction is *analog circuits*, in which devices are assembled to create components such as amplifiers. Analog circuits input and output a continuous range of voltages. *Digital circuits* such as logic gates restrict the voltages to discrete ranges, which we will use to indicate 0 and 1. In logic design, we build more complex structures, such as adders or memories, from digital circuits.

Microarchitecture links the logic and architecture levels of abstraction. The *architecture* level of abstraction describes a computer from the programmer's perspective. For example, the Intel x86 architecture used by microprocessors in most *personal computers* (*PCs*) is defined by a set of instructions and registers (memory for temporarily storing variables) that the programmer is allowed to use. Microarchitecture involves combining logic elements to execute the instructions defined by the architecture. A particular architecture can be implemented by one of many different microarchitectures with different price/performance/power trade-offs. For example, the Intel Core i7, the Intel 80486, and the AMD Athlon all implement the x86 architecture with different microarchitectures.

Moving into the software realm, the operating system handles low-level details such as accessing a hard drive or managing memory. Finally, the application software uses these facilities provided by the operating system to solve a problem for the user. Thanks to the power of abstraction, your grandmother can surf the Web without any regard for the quantum vibrations of electrons or the organization of the memory in her computer.

This book focuses on the levels of abstraction from digital circuits through computer architecture. When you are working at one level of abstraction, it is good to know something about the levels of abstraction immediately above and below where you are working. For example, a computer scientist cannot fully optimize code without understanding the architecture for which the program is being written. A device engineer cannot make wise trade-offs in transistor design without understanding the circuits in which the transistors will be used. We hope that by the time you finish reading this book, you can pick the level of abstraction appropriate to solving your problem and evaluate the impact of your design choices on other levels of abstraction.

Each chapter in this book begins with an abstraction icon indicating the focus of the chapter in deep blue, with secondary topics shown in lighter shades of blue.

1.2.2 Discipline

Discipline is the act of intentionally restricting your design choices so that you can work more productively at a higher level of abstraction. Using interchangeable parts is a familiar application of discipline. One of the first examples of interchangeable parts was in flintlock rifle manufacturing. Until the early 19th century, rifles were individually crafted by hand. Components purchased from many different craftsmen were carefully filed and fit together by a highly skilled gunmaker. The discipline of interchangeable parts revolutionized the industry. By limiting the components to a standardized set with well-defined tolerances, rifles could be assembled and repaired much faster and with less skill. The gunmaker no longer concerned himself with lower levels of abstraction such as the specific shape of an individual barrel or gunstock.

In the context of this book, the digital discipline will be very important. Digital circuits use discrete voltages, whereas analog circuits use continuous voltages. Therefore, digital circuits are a subset of analog circuits and in some sense must be capable of less than the broader class of analog circuits. However, digital circuits are much simpler to design. By limiting ourselves to digital circuits, we can easily combine components into sophisticated systems that ultimately outperform those built from analog components in many applications. For example, digital televisions, compact disks (CDs), and cell phones are replacing their analog predecessors.

1.2.3 The Three-Y's

In addition to abstraction and discipline, designers use the three "-y's" to manage complexity: hierarchy, modularity, and regularity. These principles apply to both software and hardware systems.

- *Hierarchy* involves dividing a system into modules, then further subdividing each of these modules until the pieces are easy to understand.
- *Modularity* states that the modules have well-defined functions and interfaces, so that they connect together easily without unanticipated side effects.
- *Regularity* seeks uniformity among the modules. Common modules are reused many times, reducing the number of distinct modules that must be designed.

To illustrate these "-y's" we return to the example of rifle manufacturing. A flintlock rifle was one of the most intricate objects in common use in the early 19th century. Using the principle of hierarchy, we can break it into components shown in Figure 1.2: the lock, stock, and barrel.

The barrel is the long metal tube through which the bullet is fired. The lock is the firing mechanism. And the stock is the wooden body that holds the parts together and provides a secure grip for the user. In turn, the lock contains the trigger, hammer, flint, frizzen, and pan. Each of these components could be hierarchically described in further detail.

Modularity teaches that each component should have a well-defined function and interface. A function of the stock is to mount the barrel and lock. Its interface consists of its length and the location of its mounting pins. In a modular rifle design, stocks from many different manufacturers can be used with a particular barrel as long as the stock and barrel are of the correct length and have the proper mounting mechanism. A function of the barrel is to impart spin to the bullet so that it travels more accurately. Modularity dictates that there should be no side effects: the design of the stock should not impede the function of the barrel.

Regularity teaches that interchangeable parts are a good idea. With regularity, a damaged barrel can be replaced by an identical part. The barrels can be efficiently built on an assembly line, instead of being painstakingly hand-crafted.

We will return to these principles of hierarchy, modularity, and regularity throughout the book.

Captain Meriwether Lewis of the Lewis and Clark Expedition was one of the early advocates of interchangeable parts for rifles. In 1806, he explained:

> The guns of Drewyer and Sergt. Pryor were both out of order. The first was repared with a new lock, the old one having become unfit for use; the second had the cock screw broken which was replaced by a duplicate which had been prepared for the lock at Harpers Ferry where she was manufactured. But for the precaution taken in bringing on those extra locks, and parts of locks, in addition to the ingenuity of John Shields, most of our guns would at this moment be entirely unfit for use; but fortunately for us I have it in my power here to record that they are all in good order.

See Elliott Coues, ed., *The History of the Lewis and Clark Expedition…* (4 vols), New York: Harper, 1893; reprint, 3 vols, New York: Dover, 3:817.

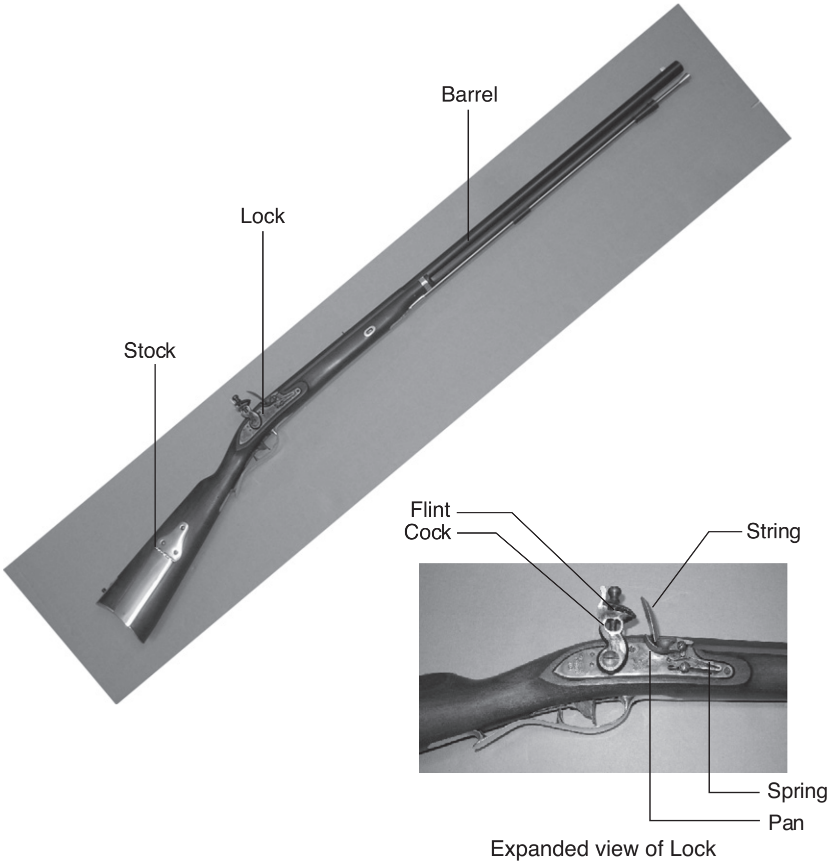


Figure 1.2 **Flintlock rifle with a close-up view of the lock** (Image by Euroarms Italia. *www.euroarms.net* © 2006.)

1.3 THE DIGITAL ABSTRACTION

Most physical variables are continuous. For example, the voltage on a wire, the frequency of an oscillation, or the position of a mass are all continuous quantities. Digital systems, on the other hand, represent information with *discrete-valued variables*—that is, variables with a finite number of distinct values.

An early digital system using variables with ten discrete values was Charles Babbage's Analytical Engine. Babbage labored from 1834 to 1871,[1] designing and attempting to build this mechanical computer. The Analytical Engine used gears with ten positions labeled 0 through 9, much like a mechanical odometer in a car. Figure 1.3 shows a prototype

Charles Babbage, 1791–1871. Attended Cambridge University and married Georgiana Whitmore in 1814. Invented the Analytical Engine, the world's first mechanical computer. Also invented the cowcatcher and the universal postage rate. Interested in lock-picking, but abhorred street musicians (image courtesy of Fourmilab Switzerland, *www.fourmilab.ch*).

---

[1] And we thought graduate school was long!

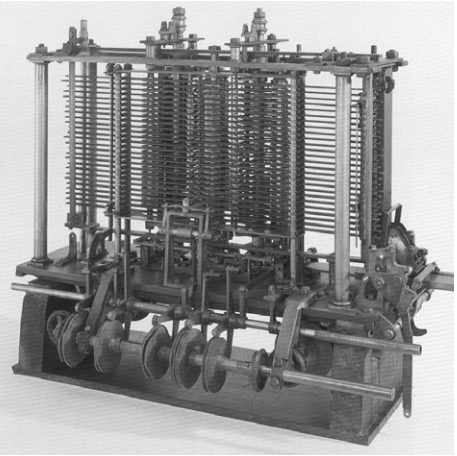


Figure 1.3 **Babbage's Analytical Engine, under construction at the time of his death in 1871** (image courtesy of Science Museum/Science and Society Picture Library)

George Boole, 1815–1864. Born to working-class parents and unable to afford a formal education, Boole taught himself mathematics and joined the faculty of Queen's College in Ireland. He wrote *An Investigation of the Laws of Thought* (1854), which introduced binary variables and the three fundamental logic operations: AND, OR, and NOT (image courtesy of the American Institute of Physics).

of the Analytical Engine, in which each row processes one digit. Babbage chose 25 rows of gears, so the machine has 25-digit precision.

Unlike Babbage's machine, most electronic computers use a binary (two-valued) representation in which a high voltage indicates a '1' and a low voltage indicates a '0', because it is easier to distinguish between two voltages than ten.

The *amount of information* D in a discrete valued variable with N distinct states is measured in units of *bits* as

$$D = \log_2 N \text{ bits} \tag{1.1}$$

A binary variable conveys $\log_2 2 = 1$ bit of information. Indeed, the word bit is short for *b*inary dig*it*. Each of Babbage's gears carried $\log_2 10 = 3.322$ bits of information because it could be in one of $2^{3.322} = 10$ unique positions. A continuous signal theoretically contains an infinite amount of information because it can take on an infinite number of values. In practice, noise and measurement error limit the information to only 10 to 16 bits for most continuous signals. If the measurement must be made rapidly, the information content is lower (e.g., 8 bits).

This book focuses on digital circuits using binary variables: 1's and 0's. George Boole developed a system of logic operating on binary variables that is now known as *Boolean logic*. Each of Boole's variables could be TRUE or FALSE. Electronic computers commonly use a positive voltage to represent '1' and zero volts to represent '0'. In this book, we will use the terms '1', TRUE, and HIGH synonymously. Similarly, we will use '0', FALSE, and LOW interchangeably.

The beauty of the *digital abstraction* is that digital designers can focus on 1's and 0's, ignoring whether the Boolean variables are physically represented with specific voltages, rotating gears, or even hydraulic fluid levels.

A computer programmer can work without needing to know the intimate details of the computer hardware. On the other hand, understanding the details of the hardware allows the programmer to optimize the software better for that specific computer.

An individual bit doesn't carry much information. In the next section, we examine how groups of bits can be used to represent numbers. In later chapters, we will also use groups of bits to represent letters and programs.

1.4 NUMBER SYSTEMS

You are accustomed to working with decimal numbers. In digital systems consisting of 1's and 0's, binary or hexadecimal numbers are often more convenient. This section introduces the various number systems that will be used throughout the rest of the book.

1.4.1 Decimal Numbers

In elementary school, you learned to count and do arithmetic in *decimal*. Just as you (probably) have ten fingers, there are ten decimal digits: 0, 1, 2, …, 9. Decimal digits are joined together to form longer decimal numbers. Each column of a decimal number has ten times the weight of the previous column. From right to left, the column weights are 1, 10, 100, 1000, and so on. Decimal numbers are referred to as *base 10*. The base is indicated by a subscript after the number to prevent confusion when working in more than one base. For example, Figure 1.4 shows how the decimal number 9742_{10} is written as the sum of each of its digits multiplied by the weight of the corresponding column.

An N-digit decimal number represents one of 10^N possibilities: 0, 1, 2, 3, …, $10^N - 1$. This is called the *range* of the number. For example, a three-digit decimal number represents one of 1000 possibilities in the range of 0 to 999.

1.4.2 Binary Numbers

Bits represent one of two values, 0 or 1, and are joined together to form *binary numbers*. Each column of a binary number has twice the weight of the previous column, so binary numbers are *base 2*. In binary, the

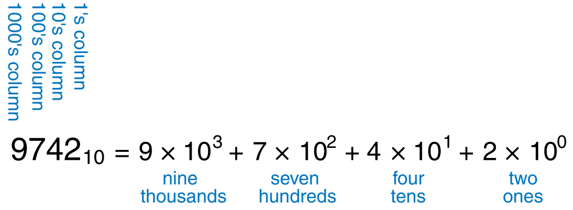


Figure 1.4 **Representation of a decimal number**

column weights (again from right to left) are 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, and so on. If you work with binary numbers often, you'll save time if you remember these powers of two up to 2^{16}.

An N-bit binary number represents one of 2^N possibilities: 0, 1, 2, 3, …, $2^N - 1$. Table 1.1 shows 1, 2, 3, and 4-bit binary numbers and their decimal equivalents.

Example 1.1 BINARY TO DECIMAL CONVERSION

Convert the binary number 10110_2 to decimal.

Solution: Figure 1.5 shows the conversion.

Table 1.1 **Binary numbers and their decimal equivalent**

1-Bit Binary Numbers	2-Bit Binary Numbers	3-Bit Binary Numbers	4-Bit Binary Numbers	Decimal Equivalents
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

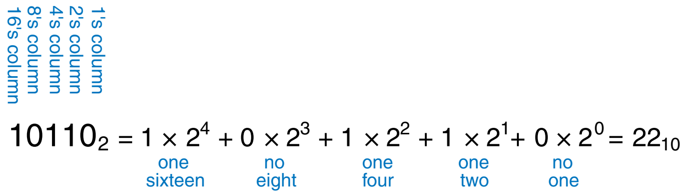


Figure 1.5 Conversion of a binary number to decimal

Example 1.2 DECIMAL TO BINARY CONVERSION

Convert the decimal number 84_{10} to binary.

Solution: Determine whether each column of the binary result has a 1 or a 0. We can do this starting at either the left or the right column.

Working from the left, start with the largest power of 2 less than or equal to the number (in this case, 64). $84 \geq 64$, so there is a 1 in the 64's column, leaving $84 - 64 = 20$. $20 < 32$, so there is a 0 in the 32's column. $20 \geq 16$, so there is a 1 in the 16's column, leaving $20 - 16 = 4$. $4 < 8$, so there is a 0 in the 8's column. $4 \geq 4$, so there is a 1 in the 4's column, leaving $4 - 4 = 0$. Thus there must be 0's in the 2's and 1's column. Putting this all together, $84_{10} = 1010100_2$.

Working from the right, repeatedly divide the number by 2. The remainder goes in each column. $84/2 = 42$, so 0 goes in the 1's column. $42/2 = 21$, so 0 goes in the 2's column. $21/2 = 10$ with a remainder of 1 going in the 4's column. $10/2 = 5$, so 0 goes in the 8's column. $5/2 = 2$ with a remainder of 1 going in the 16's column. $2/2 = 1$, so 0 goes in the 32's column. Finally $1/2 = 0$ with a remainder of 1 going in the 64's column. Again, $84_{10} = 1010100_2$.

1.4.3 Hexadecimal Numbers

Writing long binary numbers becomes tedious and prone to error. A group of four bits represents one of $2^4 = 16$ possibilities. Hence, it is sometimes more convenient to work in *base 16*, called *hexadecimal*. Hexadecimal numbers use the digits 0 to 9 along with the letters A to F, as shown in Table 1.2. Columns in base 16 have weights of 1, 16, 16^2 (or 256), 16^3 (or 4096), and so on.

"Hexadecimal," a term coined by IBM in 1963, derives from the Greek *hexi* (six) and Latin *decem* (ten). A more proper term would use the Latin *sexa* (six), but *sexadecimal* sounded too risqué.

Example 1.3 HEXADECIMAL TO BINARY AND DECIMAL CONVERSION

Convert the hexadecimal number $2ED_{16}$ to binary and to decimal.

Solution: Conversion between hexadecimal and binary is easy because each hexadecimal digit directly corresponds to four binary digits. $2_{16} = 0010_2$, $E_{16} = 1110_2$ and $D_{16} = 1101_2$, so $2ED_{16} = 001011101101_2$. Conversion to decimal requires the arithmetic shown in Figure 1.6.

Table 1.2 **Hexadecimal number system**

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

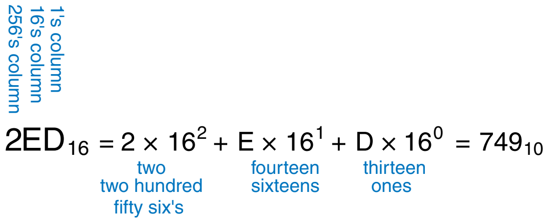

1's column
16's column
256's column
$2ED_{16} = 2 \times 16^2 + E \times 16^1 + D \times 16^0 = 749_{10}$
two
two hundred
fifty six's
fourteen
sixteens
thirteen
ones

Figure 1.6 **Conversion of a hexadecimal number to decimal**

Example 1.4 BINARY TO HEXADECIMAL CONVERSION

Convert the binary number 1111010_2 to hexadecimal.

Solution: Again, conversion is easy. Start reading from the right. The four least significant bits are $1010_2 = A_{16}$. The next bits are $111_2 = 7_{16}$. Hence $1111010_2 = 7A_{16}$.

Example 1.5 DECIMAL TO HEXADECIMAL AND BINARY CONVERSION

Convert the decimal number 333_{10} to hexadecimal and binary.

Solution: Like decimal to binary conversion, decimal to hexadecimal conversion can be done from the left or the right.

Working from the left, start with the largest power of 16 less than or equal to the number (in this case, 256). 256 goes into 333 once, so there is a 1 in the 256's column, leaving $333 - 256 = 77$. 16 goes into 77 four times, so there is a 4 in the 16's column, leaving $77 - 16 \times 4 = 13$. $13_{10} = \text{D}_{16}$, so there is a D in the 1's column. In summary, $333_{10} = 14\text{D}_{16}$. Now it is easy to convert from hexadecimal to binary, as in Example 1.3. $14\text{D}_{16} = 101001101_2$.

Working from the right, repeatedly divide the number by 16. The remainder goes in each column. $333/16 = 20$ with a remainder of $13_{10} = \text{D}_{16}$ going in the 1's column. $20/16 = 1$ with a remainder of 4 going in the 16's column. $1/16 = 0$ with a remainder of 1 going in the 256's column. Again, the result is 14D_{16}.

1.4.4 Bytes, Nibbles, and All That Jazz

A group of eight bits is called a *byte*. It represents one of $2^8 = 256$ possibilities. The size of objects stored in computer memories is customarily measured in bytes rather than bits.

A group of four bits, or half a byte, is called a *nibble*. It represents one of $2^4 = 16$ possibilities. One hexadecimal digit stores one nibble and two hexadecimal digits store one full byte. Nibbles are no longer a commonly used unit, but the term is cute.

Microprocessors handle data in chunks called *words*. The size of a word depends on the architecture of the microprocessor. When this chapter was written in 2012, most computers had 64-bit processors, indicating that they operate on 64-bit words. At the time, older computers handling 32-bit words were also widely available. Simpler microprocessors, especially those used in gadgets such as toasters, use 8- or 16-bit words.

Within a group of bits, the bit in the 1's column is called the *least significant bit* (*lsb*), and the bit at the other end is called the *most significant bit* (*msb*), as shown in Figure 1.7(a) for a 6-bit binary number. Similarly, within a word, the bytes are identified as *least significant byte* (*LSB*) through *most significant byte* (*MSB*), as shown in Figure 1.7(b) for a four-byte number written with eight hexadecimal digits.

A *microprocessor* is a processor built on a single chip. Until the 1970's, processors were too complicated to fit on one chip, so mainframe processors were built from boards containing many chips. Intel introduced the first 4-bit microprocessor, called the 4004, in 1971. Now, even the most sophisticated supercomputers are built using microprocessors. We will use the terms microprocessor and processor interchangeably throughout this book.

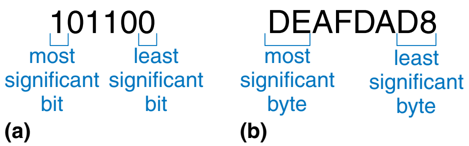


Figure 1.7 **Least and most significant bits and bytes**

By handy coincidence, $2^{10} = 1024 \approx 10^3$. Hence, the term *kilo* (Greek for thousand) indicates 2^{10}. For example, 2^{10} bytes is one kilobyte (1 KB). Similarly, *mega* (million) indicates $2^{20} \approx 10^6$, and *giga* (billion) indicates $2^{30} \approx 10^9$. If you know $2^{10} \approx 1$ thousand, $2^{20} \approx 1$ million, $2^{30} \approx 1$ billion, and remember the powers of two up to 2^9, it is easy to estimate any power of two in your head.

Example 1.6 ESTIMATING POWERS OF TWO

Find the approximate value of 2^{24} without using a calculator.

Solution: Split the exponent into a multiple of ten and the remainder.

$2^{24} = 2^{20} \times 2^4$. $2^{20} \approx 1$ million. $2^4 = 16$. So $2^{24} \approx 16$ million. Technically, $2^{24} = 16{,}777{,}216$, but 16 million is close enough for marketing purposes.

1024 bytes is called a *kilobyte* (KB). 1024 bits is called a *kilobit* (Kb or Kbit). Similarly, MB, Mb, GB, and Gb are used for millions and billions of bytes and bits. Memory capacity is usually measured in bytes. Communication speed is usually measured in bits/sec. For example, the maximum speed of a dial-up modem is usually 56 kbits/sec.

1.4.5 Binary Addition

Binary addition is much like decimal addition, but easier, as shown in Figure 1.8. As in decimal addition, if the sum of two numbers is greater than what fits in a single digit, we *carry* a 1 into the next column. Figure 1.8 compares addition of decimal and binary numbers. In the right-most column of Figure 1.8(a), $7 + 9 = 16$, which cannot fit in a single digit because it is greater than 9. So we record the 1's digit, 6, and carry the 10's digit, 1, over to the next column. Likewise, in binary, if the sum of two numbers is greater than 1, we carry the 2's digit over to the next column. For example, in the right-most column of Figure 1.8(b),

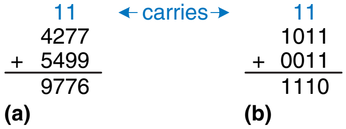


Figure 1.8 **Addition examples showing carries: (a) decimal (b) binary**

the sum $1+1=2_{10}=10_2$ cannot fit in a single binary digit. So we record the 1's digit (0) and carry the 2's digit (1) of the result to the next column. In the second column, the sum is $1+1+1=3_{10}=11_2$. Again, we record the 1's digit (1) and carry the 2's digit (1) to the next column. For obvious reasons, the bit that is carried over to the neighboring column is called the *carry bit*.

Example 1.7 BINARY ADDITION

Compute 0111_2+0101_2.

Solution: Figure 1.9 shows that the sum is 1100_2. The carries are indicated in blue. We can check our work by repeating the computation in decimal. $0111_2=7_{10}$. $0101_2=5_{10}$. The sum is $12_{10}=1100_2$.

$$\begin{array}{r} 111 \\ 0111 \\ +\ 0101 \\ \hline 1100 \end{array}$$

Figure 1.9 Binary addition example

Digital systems usually operate on a fixed number of digits. Addition is said to *overflow* if the result is too big to fit in the available digits. A 4-bit number, for example, has the range [0, 15]. 4-bit binary addition overflows if the result exceeds 15. The fifth bit is discarded, producing an incorrect result in the remaining four bits. Overflow can be detected by checking for a carry out of the most significant column.

Example 1.8 ADDITION WITH OVERFLOW

Compute 1101_2+0101_2. Does overflow occur?

Solution: Figure 1.10 shows the sum is 10010_2. This result overflows the range of a 4-bit binary number. If it must be stored as four bits, the most significant bit is discarded, leaving the incorrect result of 0010_2. If the computation had been done using numbers with five or more bits, the result 10010_2 would have been correct.

$$\begin{array}{r} 111 \\ 1101 \\ +\ 0101 \\ \hline 10010 \end{array}$$

Figure 1.10 Binary addition example with overflow

1.4.6 Signed Binary Numbers

So far, we have considered only *unsigned* binary numbers that represent positive quantities. We will often want to represent both positive and negative numbers, requiring a different binary number system. Several schemes exist to represent *signed* binary numbers; the two most widely employed are called sign/magnitude and two's complement.

Sign/Magnitude Numbers

Sign/magnitude numbers are intuitively appealing because they match our custom of writing negative numbers with a minus sign followed by the magnitude. An N-bit sign/magnitude number uses the most significant bit as the sign and the remaining $N-1$ bits as the magnitude (absolute value). A sign bit of 0 indicates positive and a sign bit of 1 indicates negative.

The $7 billion Ariane 5 rocket, launched on June 4, 1996, veered off course 40 seconds after launch, broke up, and exploded. The failure was caused when the computer controlling the rocket overflowed its 16-bit range and crashed.

The code had been extensively tested on the Ariane 4 rocket. However, the Ariane 5 had a faster engine that produced larger values for the control computer, leading to the overflow.

(Photograph courtesy of ESA/CNES/ARIANESPACE-Service Optique CS6.)

Example 1.9 SIGN/MAGNITUDE NUMBERS

Write 5 and −5 as 4-bit sign/magnitude numbers

Solution: Both numbers have a magnitude of $5_{10} = 101_2$. Thus, $5_{10} = 0101_2$ and $-5_{10} = 1101_2$.

Unfortunately, ordinary binary addition does not work for sign/magnitude numbers. For example, using ordinary addition on $-5_{10} + 5_{10}$ gives $1101_2 + 0101_2 = 10010_2$, which is nonsense.

An N-bit sign/magnitude number spans the range $[-2^{N-1} + 1, 2^{N-1} - 1]$. Sign/magnitude numbers are slightly odd in that both +0 and −0 exist. Both indicate zero. As you may expect, it can be troublesome to have two different representations for the same number.

Two's Complement Numbers

Two's complement numbers are identical to unsigned binary numbers except that the most significant bit position has a weight of -2^{N-1} instead of 2^{N-1}. They overcome the shortcomings of sign/magnitude numbers: zero has a single representation, and ordinary addition works.

In two's complement representation, zero is written as all zeros: $00\ldots000_2$. The most positive number has a 0 in the most significant position and 1's elsewhere: $01\ldots111_2 = 2^{N-1} - 1$. The most negative number has a 1 in the most significant position and 0's elsewhere: $10\ldots000_2 = -2^{N-1}$. And −1 is written as all ones: $11\ldots111_2$.

Notice that positive numbers have a 0 in the most significant position and negative numbers have a 1 in this position, so the most significant bit can be viewed as the sign bit. However, the remaining bits are interpreted differently for two's complement numbers than for sign/magnitude numbers.

The sign of a two's complement number is reversed in a process called *taking the two's complement*. The process consists of inverting all of the bits in the number, then adding 1 to the least significant bit position. This is useful to find the representation of a negative number or to determine the magnitude of a negative number.

Example 1.10 TWO'S COMPLEMENT REPRESENTATION OF A NEGATIVE NUMBER

Find the representation of -2_{10} as a 4-bit two's complement number.

Solution: Start with $+2_{10} = 0010_2$. To get -2_{10}, invert the bits and add 1. Inverting 0010_2 produces 1101_2. $1101_2 + 1 = 1110_2$. So -2_{10} is 1110_2.

Example 1.11 VALUE OF NEGATIVE TWO'S COMPLEMENT NUMBERS

Find the decimal value of the two's complement number 1001_2.

Solution: 1001_2 has a leading 1, so it must be negative. To find its magnitude, invert the bits and add 1. Inverting $1001_2 = 0110_2$. $0110_2 + 1 = 0111_2 = 7_{10}$. Hence, $1001_2 = -7_{10}$.

Two's complement numbers have the compelling advantage that addition works properly for both positive and negative numbers. Recall that when adding N-bit numbers, the carry out of the Nth bit (i.e., the $N+1^{\text{th}}$ result bit) is discarded.

Example 1.12 ADDING TWO'S COMPLEMENT NUMBERS

Compute (a) $-2_{10} + 1_{10}$ and (b) $-7_{10} + 7_{10}$ using two's complement numbers.

Solution: (a) $-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$. (b) $-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$. The fifth bit is discarded, leaving the correct 4-bit result 0000_2.

Subtraction is performed by taking the two's complement of the second number, then adding.

Example 1.13 SUBTRACTING TWO'S COMPLEMENT NUMBERS

Compute (a) $5_{10} - 3_{10}$ and (b) $3_{10} - 5_{10}$ using 4-bit two's complement numbers.

Solution: (a) $3_{10} = 0011_2$. Take its two's complement to obtain $-3_{10} = 1101_2$. Now add $5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$. Note that the carry out of the most significant position is discarded because the result is stored in four bits. (b) Take the two's complement of 5_{10} to obtain $-5_{10} = 1011$. Now add $3_{10} + (-5_{10}) = 0011_2 + 1011_2 = 1110_2 = -2_{10}$.

The two's complement of 0 is found by inverting all the bits (producing $11\ldots111_2$) and adding 1, which produces all 0's, disregarding the carry out of the most significant bit position. Hence, zero is always represented with all 0's. Unlike the sign/magnitude system, the two's complement system has no separate −0. Zero is considered positive because its sign bit is 0.

Like unsigned numbers, N-bit two's complement numbers represent one of 2^N possible values. However the values are split between positive and negative numbers. For example, a 4-bit unsigned number represents 16 values: 0 to 15. A 4-bit two's complement number also represents 16 values: −8 to 7. In general, the range of an N-bit two's complement number spans $[-2^{N-1}, 2^{N-1} - 1]$. It should make sense that there is one more negative number than positive number because there is no −0. The most negative number $10\ldots000_2 = -2^{N-1}$ is sometimes called the *weird number*. Its two's complement is found by inverting the bits (producing $01\ldots111_2$) and adding 1, which produces $10\ldots000_2$, the weird number, again. Hence, this negative number has no positive counterpart.

Adding two N-bit positive numbers or negative numbers may cause overflow if the result is greater than $2^{N-1} - 1$ or less than -2^{N-1}. Adding a positive number to a negative number never causes overflow. Unlike unsigned numbers, a carry out of the most significant column does not indicate overflow. Instead, overflow occurs if the two numbers being added have the same sign bit and the result has the opposite sign bit.

Example 1.14 ADDING TWO'S COMPLEMENT NUMBERS WITH OVERFLOW

Compute $4_{10} + 5_{10}$ using 4-bit two's complement numbers. Does the result overflow?

Solution: $4_{10} + 5_{10} = 0100_2 + 0101_2 = 1001_2 = -7_{10}$. The result overflows the range of 4-bit positive two's complement numbers, producing an incorrect negative result. If the computation had been done using five or more bits, the result $01001_2 = 9_{10}$ would have been correct.

When a two's complement number is extended to more bits, the sign bit must be copied into the most significant bit positions. This process is called *sign extension*. For example, the numbers 3 and −3 are written as 4-bit two's complement numbers 0011 and 1101, respectively. They are sign-extended to seven bits by copying the sign bit into the three new upper bits to form 0000011 and 1111101, respectively.

Comparison of Number Systems

The three most commonly used binary number systems are unsigned, two's complement, and sign/magnitude. Table 1.3 compares the range of N-bit numbers in each of these three systems. Two's complement numbers are convenient because they represent both positive and negative integers and because ordinary addition works for all numbers. Subtraction is performed by negating the second number (i.e., taking the two's

Table 1.3 **Range of *N*-bit numbers**

System	Range
Unsigned	$[0, 2^N - 1]$
Sign/Magnitude	$[-2^{N-1} + 1, 2^{N-1} - 1]$
Two's Complement	$[-2^{N-1}, 2^{N-1} - 1]$

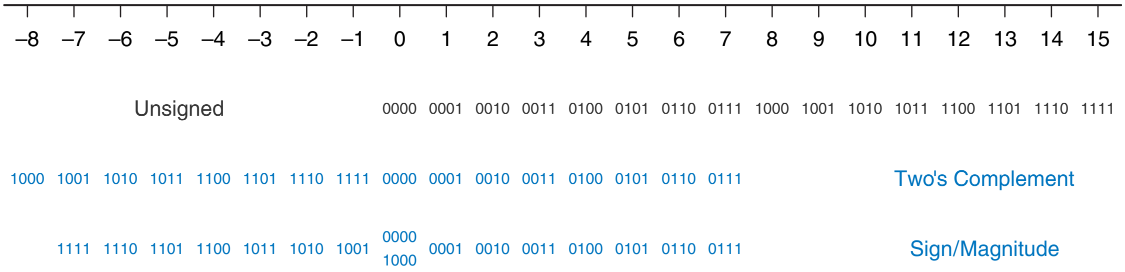

–8 –7 –6 –5 –4 –3 –2 –1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Unsigned
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
1000 1001 1010 1011 1100 1101 1110 1111 0000 0001 0010 0011 0100 0101 0110 0111
Two's Complement
1111 1110 1101 1100 1011 1010 1001 0000 1000 0001 0010 0011 0100 0101 0110 0111
Sign/Magnitude

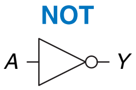
Figure 1.11 **Number line and 4-bit binary encodings**

complement), and then adding. Unless stated otherwise, assume that all signed binary numbers use two's complement representation.

Figure 1.11 shows a number line indicating the values of 4-bit numbers in each system. Unsigned numbers span the range [0, 15] in regular binary order. Two's complement numbers span the range [–8, 7]. The nonnegative numbers [0, 7] share the same encodings as unsigned numbers. The negative numbers [–8, –1] are encoded such that a larger unsigned binary value represents a number closer to 0. Notice that the weird number, 1000, represents –8 and has no positive counterpart. Sign/magnitude numbers span the range [–7, 7]. The most significant bit is the sign bit. The positive numbers [1, 7] share the same encodings as unsigned numbers. The negative numbers are symmetric but have the sign bit set. 0 is represented by both 0000 and 1000. Thus, N-bit sign/magnitude numbers represent only $2^N - 1$ integers because of the two representations for 0.

1.5 LOGIC GATES

Now that we know how to use binary variables to represent information, we explore digital systems that perform operations on these binary variables. *Logic gates* are simple digital circuits that take one or more binary inputs and produce a binary output. Logic gates are drawn with a symbol showing the input (or inputs) and the output. Inputs are usually drawn on the left (or top) and outputs on the right (or bottom). Digital designers typically use letters near the beginning of the alphabet for gate inputs and the letter Y for the gate output. The relationship between the inputs and the output can be described with a truth table or a Boolean equation. A *truth table* lists inputs on the left and the corresponding output on the right. It has one row for each possible combination of inputs. A *Boolean equation* is a mathematical expression using binary variables.



$Y = \overline{A}$

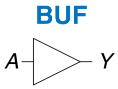
A	Y
0	1
1	0

Figure 1.12 **NOT gate**

1.5.1 NOT Gate

A *NOT gate* has one input, A, and one output, Y, as shown in Figure 1.12. The NOT gate's output is the inverse of its input. If A is FALSE, then Y is TRUE. If A is TRUE, then Y is FALSE. This relationship is summarized by the truth table and Boolean equation in the figure. The line over A in the Boolean equation is pronounced *NOT*, so $Y = \overline{A}$ is read "Y equals NOT A." The NOT gate is also called an *inverter*.

Other texts use a variety of notations for NOT, including $Y = A'$, $Y = \neg A$, $Y = !A$ or $Y = \sim A$. We will use $Y = \overline{A}$ exclusively, but don't be puzzled if you encounter another notation elsewhere.



$Y = A$

A	Y
0	0
1	1

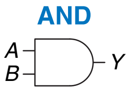
Figure 1.13 **Buffer**

1.5.2 Buffer

The other one-input logic gate is called a *buffer* and is shown in Figure 1.13. It simply copies the input to the output.

From the logical point of view, a buffer is no different from a wire, so it might seem useless. However, from the analog point of view, the buffer might have desirable characteristics such as the ability to deliver large amounts of current to a motor or the ability to quickly send its output to many gates. This is an example of why we need to consider multiple levels of abstraction to fully understand a system; the digital abstraction hides the real purpose of a buffer.

The triangle symbol indicates a buffer. A circle on the output is called a *bubble* and indicates inversion, as was seen in the NOT gate symbol of Figure 1.12.



$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 1.14 **AND gate**

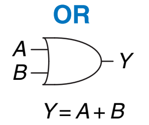
1.5.3 AND Gate

Two-input logic gates are more interesting. The *AND gate* shown in Figure 1.14 produces a TRUE output, Y, if and only if both A *and* B are TRUE. Otherwise, the output is FALSE. By convention, the inputs are listed in the order 00, 01, 10, 11, as if you were counting in binary. The Boolean equation for an AND gate can be written in several ways: $Y = A \bullet B$, $Y = AB$, or $Y = A \cap B$. The $\cap$ symbol is pronounced "intersection" and is preferred by logicians. We prefer $Y = AB$, read "Y equals A and B," because we are lazy.

According to Larry Wall, inventor of the Perl programming language, "the three principal virtues of a programmer are Laziness, Impatience, and Hubris."

1.5.4 OR Gate

The *OR gate* shown in Figure 1.15 produces a TRUE output, *Y*, if either *A* or *B* (or both) are TRUE. The Boolean equation for an OR gate is written as $Y = A + B$ or $Y = A \cup B$. The ∪ symbol is pronounced union and is preferred by logicians. Digital designers normally use the + notation, $Y = A + B$ is pronounced "*Y* equals *A* or *B*."

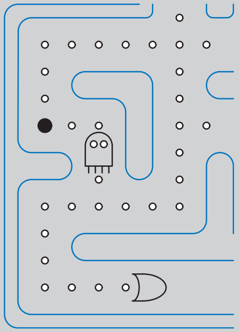


$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

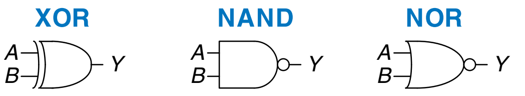
Figure 1.15 **OR gate**

A silly way to remember the OR symbol is that its input side is curved like Pacman's mouth, so the gate is hungry and willing to eat any TRUE inputs it can find!



1.5.5 Other Two-Input Gates

Figure 1.16 shows other common two-input logic gates. *XOR* (exclusive OR, pronounced "ex-OR") is TRUE if *A* or *B*, but not both, are TRUE. Any gate can be followed by a bubble to invert its operation. The *NAND gate* performs NOT AND. Its output is TRUE unless both inputs are TRUE. The *NOR gate* performs NOT OR. Its output is TRUE if neither *A* nor *B* is TRUE. An *N*-input XOR gate is sometimes called a *parity* gate and produces a TRUE output if an odd number of inputs are TRUE. As with two-input gates, the input combinations in the truth table are listed in counting order.



XOR: $Y = A \oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

NAND: $Y = \overline{AB}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR: $Y = \overline{A + B}$

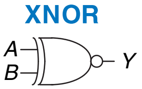
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Figure 1.16 **More two-input logic gates**

Example 1.15 XNOR GATE

Figure 1.17 shows the symbol and Boolean equation for a two-input *XNOR gate* that performs the inverse of an XOR. Complete the truth table.

Solution: Figure 1.18 shows the truth table. The XNOR output is TRUE if both inputs are FALSE or both inputs are TRUE. The two-input XNOR gate is sometimes called an *equality* gate because its output is TRUE when the inputs are equal.



$Y = \overline{A \oplus B}$

A	B	Y
0	0	
0	1	
1	0	
1	1	

Figure 1.17 **XNOR gate**

1.5.6 Multiple-Input Gates

Many Boolean functions of three or more inputs exist. The most common are AND, OR, XOR, NAND, NOR, and XNOR. An *N*-input AND gate produces a TRUE output when all *N* inputs are TRUE. An *N*-input OR gate produces a TRUE output when at least one input is TRUE.

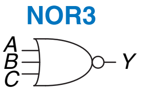
A	*B*	*Y*
0	0	1
0	1	0
1	0	0
1	1	1

Figure 1.18 **XNOR truth table**

Example 1.16 THREE-INPUT NOR GATE

Figure 1.19 shows the symbol and Boolean equation for a three-input NOR gate. Complete the truth table.

Solution: Figure 1.20 shows the truth table. The output is TRUE only if none of the inputs are TRUE.



$Y = \overline{A + B + C}$

A	*B*	*C*	*Y*
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Figure 1.19 **Three-input NOR gate**

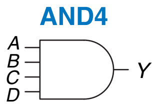
A	*B*	*C*	*Y*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Figure 1.20 **Three-input NOR truth table**

Example 1.17 FOUR-INPUT AND GATE

Figure 1.21 shows the symbol and Boolean equation for a four-input AND gate. Create a truth table.

Solution: Figure 1.22 shows the truth table. The output is TRUE only if all of the inputs are TRUE.



$Y = ABCD$

Figure 1.21 **Four-input AND gate**

1.6 BENEATH THE DIGITAL ABSTRACTION

A digital system uses discrete-valued variables. However, the variables are represented by continuous physical quantities such as the voltage on a wire, the position of a gear, or the level of fluid in a cylinder. Hence, the designer must choose a way to relate the continuous value to the discrete value.

For example, consider representing a binary signal *A* with a voltage on a wire. Let 0 volts (V) indicate $A = 0$ and 5 V indicate $A = 1$. Any real system must tolerate some noise, so 4.97 V probably ought to be interpreted as $A = 1$ as well. But what about 4.3 V? Or 2.8 V? Or 2.500000 V?

1.6.1 Supply Voltage

Suppose the lowest voltage in the system is 0 V, also called *ground* or GND. The highest voltage in the system comes from the power supply and is usually called V_{DD}. In 1970's and 1980's technology, V_{DD} was generally 5 V. As chips have progressed to smaller transistors, V_{DD} has dropped to 3.3 V, 2.5 V, 1.8 V, 1.5 V, 1.2 V, or even lower to save power and avoid overloading the transistors.

1.6.2 Logic Levels

The mapping of a continuous variable onto a discrete binary variable is done by defining *logic levels*, as shown in Figure 1.23. The first gate is called the *driver* and the second gate is called the *receiver*. The output of the driver is connected to the input of the receiver. The driver produces a LOW (0) output in the range of 0 to V_{OL} or a HIGH (1) output in the range of V_{OH} to V_{DD}. If the receiver gets an input in the range of 0 to V_{IL}, it will consider the input to be LOW. If the receiver gets an input in the range of V_{IH} to V_{DD}, it will consider the input to be HIGH. If, for some reason such as noise or faulty components, the receiver's input should fall in the *forbidden zone* between V_{IL} and V_{IH}, the behavior of the gate is unpredictable. V_{OH},V_{OL}, V_{IH}, and V_{IL} are called the output and input high and low logic levels.

1.6.3 Noise Margins

If the output of the driver is to be correctly interpreted at the input of the receiver, we must choose $V_{OL} < V_{IL}$ and $V_{OH} > V_{IH}$. Thus, even if the output of the driver is contaminated by some noise, the input of the receiver will still detect the correct logic level. The *noise margin* is the amount of noise that could be added to a worst-case output such that the signal can still be interpreted as a valid input. As can be seen in Figure 1.23, the low and high noise margins are, respectively

$$NM_L = V_{IL} - V_{OL} \tag{1.2}$$

$$NM_H = V_{OH} - V_{IH} \tag{1.3}$$

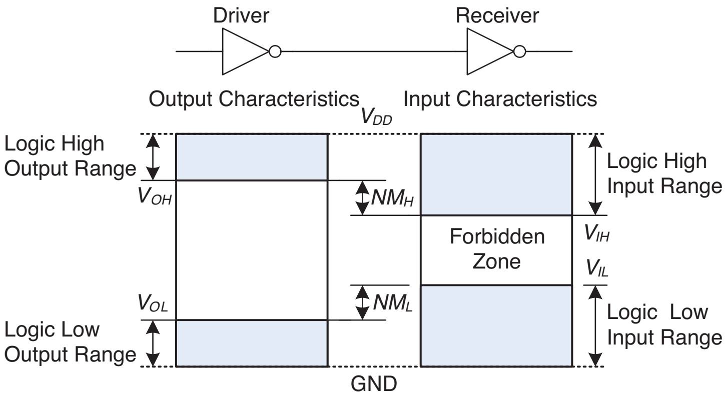


Figure 1.23 **Logic levels and noise margins**

Example 1.18 CALCULATING NOISE MARGINS

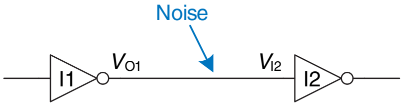
Consider the inverter circuit of Figure 1.24. V_{O1} is the output voltage of inverter I1, and V_{I2} is the input voltage of inverter I2. Both inverters have the following characteristics: $V_{DD} = 5$ V, $V_{IL} = 1.35$ V, $V_{IH} = 3.15$ V, $V_{OL} = 0.33$ V, and $V_{OH} = 3.84$ V. What are the inverter low and high noise margins? Can the circuit tolerate 1 V of noise between V_{O1} and V_{I2}?

A	*C*	*B*	*D*	*Y*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 1.22 **Four-input AND truth table**

V_{DD} stands for the voltage on the *drain* of a metal-oxide-semiconductor transistor, used to build most modern chips. The power supply voltage is also sometimes called V_{CC}, standing for the voltage on the *collector* of a bipolar junction transistor used to build chips in an older technology. Ground is sometimes called V_{SS} because it is the voltage on the *source* of a metal-oxide-semiconductor transistor. See Section 1.7 for more information on transistors.

Figure 1.24 **Inverter circuit**



Solution: The inverter noise margins are: $NM_L = V_{IL} - V_{OL} = (1.35\text{ V} - 0.33\text{ V}) = 1.02\text{ V}$, $NM_H = V_{OH} - V_{IH} = (3.84\text{ V} - 3.15\text{ V}) = 0.69\text{ V}$. The circuit can tolerate 1 V of noise when the output is LOW ($NM_L = 1.02$ V) but not when the output is HIGH ($NM_H = 0.69$ V). For example, suppose the driver, I1, outputs its worst-case HIGH value, $V_{O1} = V_{OH} = 3.84$ V. If noise causes the voltage to droop by 1 V before reaching the input of the receiver, $V_{I2} = (3.84\text{ V} - 1\text{ V}) = 2.84$ V. This is less than the acceptable input HIGH value, $V_{IH} = 3.15$ V, so the receiver may not sense a proper HIGH input.

DC indicates behavior when an input voltage is held constant or changes slowly enough for the rest of the system to keep up. The term's historical root comes from *direct current*, a method of transmitting power across a line with a constant voltage. In contrast, the *transient response* of a circuit is the behavior when an input voltage changes rapidly. Section 2.9 explores transient response further.

1.6.4 DC Transfer Characteristics

To understand the limits of the digital abstraction, we must delve into the analog behavior of a gate. The DC *transfer characteristics* of a gate describe the output voltage as a function of the input voltage when the input is changed slowly enough that the output can keep up. They are called transfer characteristics because they describe the relationship between input and output voltages.

An ideal inverter would have an abrupt switching threshold at $V_{DD}/2$, as shown in Figure 1.25(a). For $V(A) < V_{DD}/2$, $V(Y) = V_{DD}$. For $V(A) > V_{DD}/2$, $V(Y) = 0$. In such a case, $V_{IH} = V_{IL} = V_{DD}/2$. $V_{OH} = V_{DD}$ and $V_{OL} = 0$.

A real inverter changes more gradually between the extremes, as shown in Figure 1.25(b). When the input voltage $V(A)$ is 0, the output voltage $V(Y) = V_{DD}$. When $V(A) = V_{DD}$, $V(Y) = 0$. However, the transition between these endpoints is smooth and may not be centered at exactly $V_{DD}/2$. This raises the question of how to define the logic levels.

A reasonable place to choose the logic levels is where the slope of the transfer characteristic $dV(Y) / dV(A)$ is -1. These two points are called the *unity gain points*. Choosing logic levels at the unity gain points usually maximizes the noise margins. If V_{IL} were reduced, V_{OH} would only increase by a small amount. But if V_{IL} were increased, V_{OH} would drop precipitously.

1.6.5 The Static Discipline

To avoid inputs falling into the forbidden zone, digital logic gates are designed to conform to the *static discipline*. The static discipline requires that, given logically valid inputs, every circuit element will produce logically valid outputs.

By conforming to the static discipline, digital designers sacrifice the freedom of using arbitrary analog circuit elements in return for the simplicity and robustness of digital circuits. They raise the level of abstraction

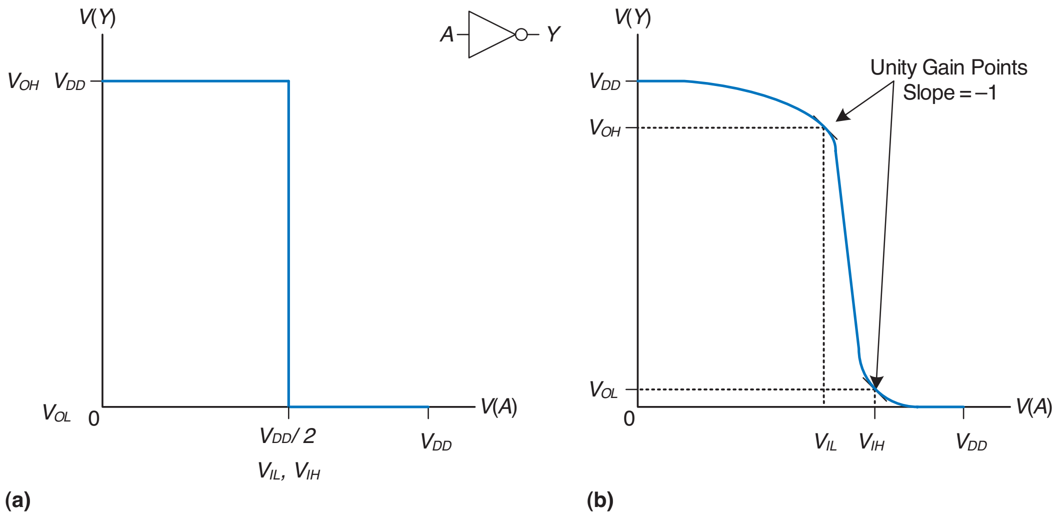


Figure 1.25 DC transfer characteristics and logic levels

from analog to digital, increasing design productivity by hiding needless detail.

The choice of V_{DD} and logic levels is arbitrary, but all gates that communicate must have compatible logic levels. Therefore, gates are grouped into *logic families* such that all gates in a logic family obey the static discipline when used with other gates in the family. Logic gates in the same logic family snap together like Legos in that they use consistent power supply voltages and logic levels.

Four major logic families that predominated from the 1970's through the 1990's are Transistor-Transistor Logic (TTL), Complementary Metal-Oxide-Semiconductor Logic (CMOS, pronounced sea-moss), Low Voltage TTL Logic (LVTTL), and Low Voltage CMOS Logic (LVCMOS). Their logic levels are compared in Table 1.4. Since then, logic families have balkanized with a proliferation of even lower power supply voltages. Appendix A.6 revisits popular logic families in more detail.

Table 1.4 Logic levels of 5 V and 3.3 V logic families

Logic Family	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5 (4.75–5.25)	0.8	2.0	0.4	2.4
CMOS	5 (4.5–6)	1.35	3.15	0.33	3.84
LVTTL	3.3 (3–3.6)	0.8	2.0	0.4	2.4
LVCMOS	3.3 (3–3.6)	0.9	1.8	0.36	2.7

Table 1.5 **Compatibility of logic families**

		Receiver			
		TTL	CMOS	LVTTL	LVCMOS
Driver	TTL	OK	NO: $V_{OH} < V_{IH}$	MAYBE[a]	MAYBE[a]
	CMOS	OK	OK	MAYBE[a]	MAYBE[a]
	LVTTL	OK	NO: $V_{OH} < V_{IH}$	OK	OK
	LVCMOS	OK	NO: $V_{OH} < V_{IH}$	OK	OK

[a] As long as a 5 V HIGH level does not damage the receiver input

---

Example 1.19 LOGIC FAMILY COMPATIBILITY

Which of the logic families in Table 1.4 can communicate with each other reliably?

Solution: Table 1.5 lists which logic families have compatible logic levels. Note that a 5 V logic family such as TTL or CMOS may produce an output voltage as HIGH as 5 V. If this 5 V signal drives the input of a 3.3 V logic family such as LVTTL or LVCMOS, it can damage the receiver, unless the receiver is specially designed to be "5-volt compatible."

1.7 CMOS TRANSISTORS*

This section and other sections marked with a * are optional and are not necessary to understand the main flow of the book.

Babbage's Analytical Engine was built from gears, and early electrical computers used relays or vacuum tubes. Modern computers use transistors because they are cheap, small, and reliable. *Transistors* are electrically controlled switches that turn ON or OFF when a voltage or current is applied to a control terminal. The two main types of transistors are *bipolar junction transistors* and *metal-oxide-semiconductor field effect transistors* (*MOSFETs* or *MOS transistors*, pronounced "moss-fets" or "M-O-S", respectively).

In 1958, Jack Kilby at Texas Instruments built the first integrated circuit containing two transistors. In 1959, Robert Noyce at Fairchild Semiconductor patented a method of interconnecting multiple transistors on a single silicon chip. At the time, transistors cost about $10 each.

Thanks to more than three decades of unprecedented manufacturing advances, engineers can now pack roughly one billion MOSFETs onto a 1 cm^2 chip of silicon, and these transistors cost less than 10 microcents apiece. The capacity and cost continue to improve by an order of magnitude every 8 years or so. MOSFETs are now the building blocks of almost all

Robert Noyce, 1927–1990. Born in Burlington, Iowa. Received a B.A. in physics from Grinnell College and a Ph.D. in physics from MIT. Nicknamed "Mayor of Silicon Valley" for his profound influence on the industry.

Cofounded Fairchild Semiconductor in 1957 and Intel in 1968. Coinvented the integrated circuit. Many engineers from his teams went on to found other seminal semiconductor companies (© 2006, Intel Corporation. Reproduced by permission).

digital systems. In this section, we will peer beneath the digital abstraction to see how logic gates are built from MOSFETs.

1.7.1 Semiconductors

MOS transistors are built from silicon, the predominant atom in rock and sand. Silicon (Si) is a group IV atom, so it has four electrons in its valence shell and forms bonds with four adjacent atoms, resulting in a crystalline *lattice*. Figure 1.26(a) shows the lattice in two dimensions for ease of drawing, but remember that the lattice actually forms a cubic crystal. In the figure, a line represents a covalent bond. By itself, silicon is a poor conductor because all the electrons are tied up in covalent bonds. However, it becomes a better conductor when small amounts of impurities, called *dopant* atoms, are carefully added. If a group V dopant such as arsenic (As) is added, the dopant atoms have an extra electron that is not involved in the bonds. The electron can easily move about the lattice, leaving an ionized dopant atom (As^+) behind, as shown in Figure 1.26(b). The electron carries a negative charge, so we call arsenic an *n-type* dopant. On the other hand, if a group III dopant such as boron (B) is added, the dopant atoms are missing an electron, as shown in Figure 1.26(c). This missing electron is called a *hole*. An electron from a neighboring silicon atom may move over to fill the missing bond, forming an ionized dopant atom (B^-) and leaving a hole at the neighboring silicon atom. In a similar fashion, the hole can migrate around the lattice. The hole is a lack of negative charge, so it acts like a positively charged particle. Hence, we call boron a *p-type* dopant. Because the conductivity of silicon changes over many orders of magnitude depending on the concentration of dopants, silicon is called a *semiconductor*.

1.7.2 Diodes

The junction between p-type and n-type silicon is called a *diode*. The p-type region is called the *anode* and the n-type region is called the *cathode*, as illustrated in Figure 1.27. When the voltage on the anode rises above the voltage on the cathode, the diode is *forward biased*, and current

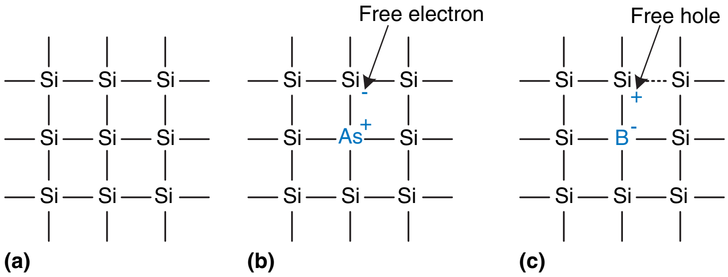


Figure 1.26 **Silicon lattice and dopant atoms**

flows through the diode from the anode to the cathode. But when the anode voltage is lower than the voltage on the cathode, the diode is *reverse biased*, and no current flows. The diode symbol intuitively shows that current only flows in one direction.

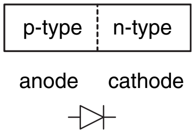


Figure 1.27 **The p-n junction diode structure and symbol**

1.7.3 Capacitors

A *capacitor* consists of two conductors separated by an insulator. When a voltage V is applied to one of the conductors, the conductor accumulates electric *charge* Q and the other conductor accumulates the opposite charge $-Q$. The *capacitance* C of the capacitor is the ratio of charge to voltage: $C = Q/V$. The capacitance is proportional to the size of the conductors and inversely proportional to the distance between them. The symbol for a capacitor is shown in Figure 1.28.

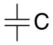


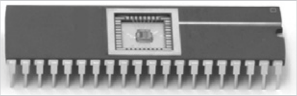
Figure 1.28 **Capacitor symbol**

Capacitance is important because charging or discharging a conductor takes time and energy. More capacitance means that a circuit will be slower and require more energy to operate. Speed and energy will be discussed throughout this book.

Technicians in an Intel clean room wear Gore-Tex bunny suits to prevent particulates from their hair, skin, and clothing from contaminating the microscopic transistors on silicon wafers (© 2006, Intel Corporation. Reproduced by permission).

1.7.4 nMOS and pMOS Transistors

A MOSFET is a sandwich of several layers of conducting and insulating materials. MOSFETs are built on thin flat *wafers* of silicon of about 15 to 30 cm in diameter. The manufacturing process begins with a bare wafer. The process involves a sequence of steps in which dopants are implanted into the silicon, thin films of silicon dioxide and silicon are grown, and metal is deposited. Between each step, the wafer is *patterned* so that the materials appear only where they are desired. Because transistors are a fraction of a micron[2] in length and the entire wafer is processed at once, it is inexpensive to manufacture billions of transistors at a time. Once processing is complete, the wafer is cut into rectangles called *chips* or *dice* that contain thousands, millions, or even billions of transistors. The chip is tested, then placed in a plastic or ceramic *package* with metal pins to connect it to a circuit board.

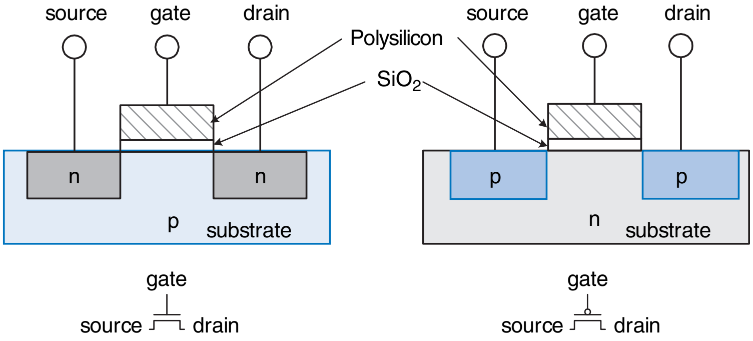


A 40-pin dual-inline package (DIP) contains a small chip (scarcely visible) in the center that is connected to 40 metal pins, 20 on a side, by gold wires thinner than a strand of hair (photograph by Kevin Mapp. © Harvey Mudd College).

The MOSFET sandwich consists of a conducting layer called the *gate* on top of an insulating layer of *silicon dioxide* (SiO_2) on top of the silicon wafer, called the *substrate*. Historically, the gate was constructed from metal, hence the name metal-oxide-semiconductor. Modern manufacturing processes use polycrystalline silicon for the gate because it does not melt during subsequent high-temperature processing steps. Silicon dioxide is better known as glass and is often simply called *oxide* in the semiconductor industry. The metal-oxide-semiconductor sandwich forms a capacitor, in which a thin layer of insulating oxide called a *dielectric* separates the metal and semiconductor plates.

---

[2] 1 μm = 1 micron = 10^{-6} m.



(a) nMOS **(b)** pMOS

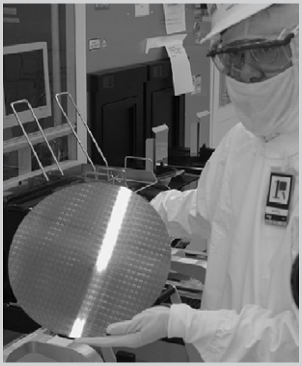
Figure 1.29 **nMOS and pMOS transistors**

There are two flavors of MOSFETs: nMOS and pMOS (pronounced "n-moss" and "p-moss"). Figure 1.29 shows cross-sections of each type, made by sawing through a wafer and looking at it from the side. The n-type transistors, called *nMOS*, have regions of n-type dopants adjacent to the gate called the *source* and the *drain* and are built on a p-type semiconductor substrate. The *pMOS* transistors are just the opposite, consisting of p-type source and drain regions in an n-type *substrate*.

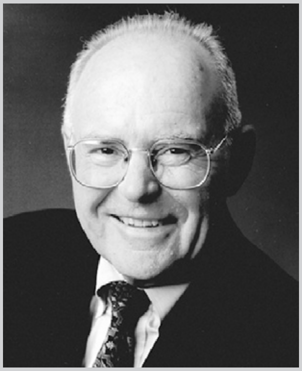
A MOSFET behaves as a voltage-controlled switch in which the gate voltage creates an electric field that turns ON or OFF a connection between the source and drain. The term *field effect transistor* comes from this principle of operation. Let us start by exploring the operation of an nMOS transistor.

The substrate of an nMOS transistor is normally tied to GND, the lowest voltage in the system. First, consider the situation when the gate is also at 0 V, as shown in Figure 1.30(a). The diodes between the source or drain and the substrate are reverse biased because the source or drain voltage is nonnegative. Hence, there is no path for current to flow between the source and drain, so the transistor is OFF. Now, consider when the gate is raised to V_{DD}, as shown in Figure 1.30(b). When a positive voltage is applied to the top plate of a capacitor, it establishes an electric field that attracts positive charge on the top plate and negative charge to the bottom plate. If the voltage is sufficiently large, so much negative charge is attracted to the underside of the gate that the region *inverts* from p-type to effectively become n-type. This inverted region is called the *channel*. Now the transistor has a continuous path from the n-type source through the n-type channel to the n-type drain, so electrons can flow from source to drain. The transistor is ON. The gate voltage required to turn on a transistor is called the *threshold voltage*, V_t, and is typically 0.3 to 0.7 V.

The source and drain terminals are physically symmetric. However, we say that charge flows from the source to the drain. In an nMOS transistor, the charge is carried by electrons, which flow from negative voltage to positive voltage. In a pMOS transistor, the charge is carried by holes, which flow from positive voltage to negative voltage. If we draw schematics with the most positive voltage at the top and the most negative at the bottom, the source of (negative) charges in an nMOS transistor is the bottom terminal and the source of (positive) charges in a pMOS transistor is the top terminal.



A technician holds a 12-inch wafer containing hundreds of microprocessor chips (© 2006, Intel Corporation. Reproduced by permission).



Gordon Moore, 1929–. Born in San Francisco. Received a B.S. in chemistry from UC Berkeley and a Ph.D. in chemistry and physics from Caltech. Cofounded Intel in 1968 with Robert Noyce. Observed in 1965 that the number of transistors on a computer chip doubles every year. This trend has become known as *Moore's Law*. Since 1975, transistor counts have doubled every two years.

A corollary of Moore's Law is that microprocessor performance doubles every 18 to 24 months. Semiconductor sales have also increased exponentially. Unfortunately, power consumption has increased exponentially as well (© 2006, Intel Corporation. Reproduced by permission).

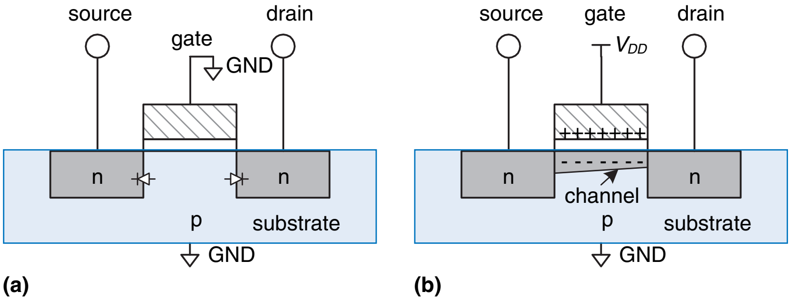


Figure 1.30 **nMOS transistor operation**

pMOS transistors work in just the opposite fashion, as might be guessed from the bubble on their symbol shown in Figure 1.31. The substrate is tied to V_{DD}. When the gate is also at V_{DD}, the pMOS transistor is OFF. When the gate is at GND, the channel inverts to p-type and the pMOS transistor is ON.

Unfortunately, MOSFETs are not perfect switches. In particular, nMOS transistors pass 0's well but pass 1's poorly. Specifically, when the gate of an nMOS transistor is at V_{DD}, the drain will only swing between 0 and $V_{DD} - V_t$. Similarly, pMOS transistors pass 1's well but 0's poorly. However, we will see that it is possible to build logic gates that use transistors only in their good mode.

nMOS transistors need a p-type substrate, and pMOS transistors need an n-type substrate. To build both flavors of transistors on the same chip, manufacturing processes typically start with a p-type wafer, then implant n-type regions called *wells* where the pMOS transistors should go. These processes that provide both flavors of transistors are called Complementary MOS or CMOS. CMOS processes are used to build the vast majority of all transistors fabricated today.

In summary, CMOS processes give us two types of electrically controlled switches, as shown in Figure 1.31. The voltage at the gate (*g*) regulates the flow of current between the source (*s*) and drain (*d*). nMOS transistors are OFF when the gate is 0 and ON when the gate is 1. pMOS

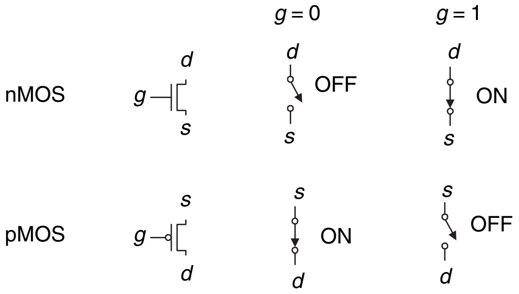


Figure 1.31 **Switch models of MOSFETs**

transistors are just the opposite: ON when the gate is 0 and OFF when the gate is 1.

1.7.5 CMOS NOT Gate

Figure 1.32 shows a schematic of a NOT gate built with CMOS transistors. The triangle indicates GND, and the flat bar indicates V_{DD}; these labels will be omitted from future schematics. The nMOS transistor, N1, is connected between GND and the Y output. The pMOS transistor, P1, is connected between V_{DD} and the Y output. Both transistor gates are controlled by the input, A.

If $A = 0$, N1 is OFF and P1 is ON. Hence, Y is connected to V_{DD} but not to GND, and is pulled up to a logic 1. P1 passes a good 1. If $A = 1$, N1 is ON and P1 is OFF, and Y is pulled down to a logic 0. N1 passes a good 0. Checking against the truth table in Figure 1.12, we see that the circuit is indeed a NOT gate.

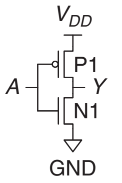


Figure 1.32 **NOT gate schematic**

1.7.6 Other CMOS Logic Gates

Figure 1.33 shows a schematic of a two-input NAND gate. In schematic diagrams, wires are always joined at three-way junctions. They are joined at four-way junctions only if a dot is shown. The nMOS transistors N1 and N2 are connected in series; both nMOS transistors must be ON to pull the output down to GND. The pMOS transistors P1 and P2 are in parallel; only one pMOS transistor must be ON to pull the output up to V_{DD}. Table 1.6 lists the operation of the pull-down and pull-up networks and the state of the output, demonstrating that the gate does function as a NAND. For example, when $A = 1$ and $B = 0$, N1 is ON, but N2 is OFF, blocking the path from Y to GND. P1 is OFF, but P2 is ON, creating a path from V_{DD} to Y. Therefore, Y is pulled up to 1.

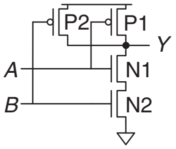


Figure 1.33 **Two-input NAND gate schematic**

Figure 1.34 shows the general form used to construct any inverting logic gate, such as NOT, NAND, or NOR. nMOS transistors are good at passing 0's, so a pull-down network of nMOS transistors is placed between the output and GND to pull the output down to 0. pMOS transistors are

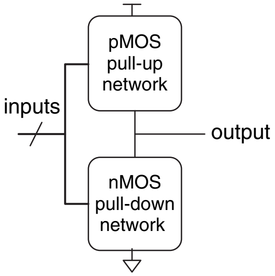


Figure 1.34 **General form of an inverting logic gate**

Table 1.6 **NAND gate operation**

A	B	Pull-Down Network	Pull-Up Network	Y
0	0	OFF	ON	1
0	1	OFF	ON	1
1	0	OFF	ON	1
1	1	ON	OFF	0

good at passing 1's, so a pull-up network of pMOS transistors is placed between the output and V_{DD} to pull the output up to 1. The networks may consist of transistors in series or in parallel. When transistors are in parallel, the network is ON if either transistor is ON. When transistors are in series, the network is ON only if both transistors are ON. The slash across the input wire indicates that the gate may receive multiple inputs.

If both the pull-up and pull-down networks were ON simultaneously, a *short circuit* would exist between V_{DD} and GND. The output of the gate might be in the forbidden zone and the transistors would consume large amounts of power, possibly enough to burn out. On the other hand, if both the pull-up and pull-down networks were OFF simultaneously, the output would be connected to neither V_{DD} nor GND. We say that the output *floats*. Its value is again undefined. Floating outputs are usually undesirable, but in Section 2.6 we will see how they can occasionally be used to the designer's advantage.

In a properly functioning logic gate, one of the networks should be ON and the other OFF at any given time, so that the output is pulled HIGH or LOW but not shorted or floating. We can guarantee this by using the rule of *conduction complements*. When nMOS transistors are in series, the pMOS transistors must be in parallel. When nMOS transistors are in parallel, the pMOS transistors must be in series.

Experienced designers claim that electronic devices operate because they contain *magic smoke*. They confirm this theory with the observation that if the magic smoke is ever let out of the device, it ceases to work.

Example 1.20 THREE-INPUT NAND SCHEMATIC

Draw a schematic for a three-input NAND gate using CMOS transistors.

Solution: The NAND gate should produce a 0 output only when all three inputs are 1. Hence, the pull-down network should have three nMOS transistors in series. By the conduction complements rule, the pMOS transistors must be in parallel. Such a gate is shown in Figure 1.35; you can verify the function by checking that it has the correct truth table.

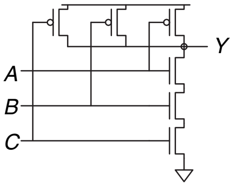


Figure 1.35 **Three-input NAND gate schematic**

Example 1.21 TWO-INPUT NOR SCHEMATIC

Draw a schematic for a two-input NOR gate using CMOS transistors.

Solution: The NOR gate should produce a 0 output if either input is 1. Hence, the pull-down network should have two nMOS transistors in parallel. By the conduction complements rule, the pMOS transistors must be in series. Such a gate is shown in Figure 1.36.

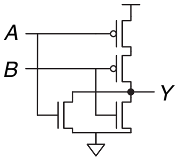


Figure 1.36 **Two-input NOR gate schematic**

Example 1.22 TWO-INPUT AND SCHEMATIC

Draw a schematic for a two-input AND gate.

Solution: It is impossible to build an AND gate with a single CMOS gate. However, building NAND and NOT gates is easy. Thus, the best way to build an AND gate using CMOS transistors is to use a NAND followed by a NOT, as shown in Figure 1.37.

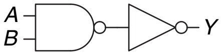


Figure 1.37 Two-input AND gate schematic

1.7.7 Transmission Gates

At times, designers find it convenient to use an ideal switch that can pass both 0 and 1 well. Recall that nMOS transistors are good at passing 0 and pMOS transistors are good at passing 1, so the parallel combination of the two passes both values well. Figure 1.38 shows such a circuit, called a *transmission gate* or *pass gate*. The two sides of the switch are called A and B because a switch is bidirectional and has no preferred input or output side. The control signals are called *enables*, EN and $\overline{EN}$. When $EN = 0$ and $\overline{EN} = 1$, both transistors are OFF. Hence, the transmission gate is OFF or disabled, so A and B are not connected. When $EN = 1$ and $\overline{EN} = 0$, the transmission gate is ON or enabled, and any logic value can flow between A and B.

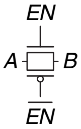


Figure 1.38 Transmission gate

1.7.8 Pseudo-nMOS Logic

An N-input CMOS NOR gate uses N nMOS transistors in parallel and N pMOS transistors in series. Transistors in series are slower than transistors in parallel, just as resistors in series have more resistance than resistors in parallel. Moreover, pMOS transistors are slower than nMOS transistors because holes cannot move around the silicon lattice as fast as electrons. Therefore the parallel nMOS transistors are fast and the series pMOS transistors are slow, especially when many are in series.

Pseudo-nMOS logic replaces the slow stack of pMOS transistors with a single weak pMOS transistor that is always ON, as shown in Figure 1.39. This pMOS transistor is often called a *weak pull-up*. The physical dimensions of the pMOS transistor are selected so that the pMOS transistor will pull the output Y HIGH weakly—that is, only if none of the nMOS transistors are ON. But if any nMOS transistor is ON, it overpowers the weak pull-up and pulls Y down close enough to GND to produce a logic 0.

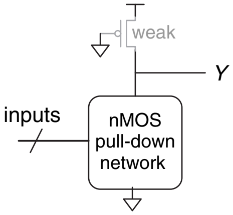


Figure 1.39 Generic pseudo-nMOS gate

The advantage of pseudo-nMOS logic is that it can be used to build fast NOR gates with many inputs. For example, Figure 1.40 shows a pseudo-nMOS four-input NOR. Pseudo-nMOS gates are useful for certain memory and logic arrays discussed in Chapter 5. The disadvantage is that a short circuit exists between V_{DD} and GND when the output is LOW; the weak pMOS and nMOS transistors are both ON. The short circuit draws continuous power, so pseudo-nMOS logic must be used sparingly.

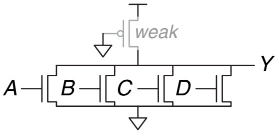


Figure 1.40 Pseudo-nMOS four-input NOR gate

Pseudo-nMOS gates got their name from the 1970's, when manufacturing processes only had nMOS transistors. A weak nMOS transistor was used to pull the output HIGH because pMOS transistors were not available.

1.8 POWER CONSUMPTION*

Power consumption is the amount of energy used per unit time. Power consumption is of great importance in digital systems. The battery life of portable systems such as cell phones and laptop computers is limited by power consumption. Power is also significant for systems that are plugged in, because electricity costs money and because the system will overheat if it draws too much power.

Digital systems draw both *dynamic* and *static* power. Dynamic power is the power used to charge capacitance as signals change between 0 and 1. Static power is the power used even when signals do not change and the system is idle.

Logic gates and the wires that connect them have capacitance. The energy drawn from the power supply to charge a capacitance C to voltage V_{DD} is CV_{DD}^2. If the voltage on the capacitor switches at frequency f (i.e., f times per second), it charges the capacitor $f/2$ times and discharges it $f/2$ times per second. Discharging does not draw energy from the power supply, so the dynamic power consumption is

$$P_{\text{dynamic}} = \frac{1}{2} C V_{DD}^2 f \tag{1.4}$$

Electrical systems draw some current even when they are idle. When transistors are OFF, they leak a small amount of current. Some circuits, such as the pseudo-nMOS gate discussed in Section 1.7.8, have a path from V_{DD} to GND through which current flows continuously. The total static current, I_{DD}, is also called the *leakage current* or the *quiescent supply current* flowing between V_{DD} and GND. The static power consumption is proportional to this static current:

$$P_{\text{static}} = I_{DD} V_{DD} \tag{1.5}$$

Example 1.23 POWER CONSUMPTION

A particular cell phone has a 6 watt-hour (W-hr) battery and operates at 1.2 V. Suppose that, when it is in use, the cell phone operates at 300 MHz and the average amount of capacitance in the chip switching at any given time is 10 nF (10^{-8} Farads). When in use, it also broadcasts 3 W of power out of its antenna. When the phone is not in use, the dynamic power drops to almost zero because the signal processing is turned off. But the phone also draws 40 mA of quiescent current whether it is in use or not. Determine the battery life of the phone (a) if it is not being used, and (b) if it is being used continuously.

Solution: The static power is $P_{static} = (0.040\ \text{A})(1.2\ \text{V}) = 48$ mW. (a) If the phone is not being used, this is the only power consumption, so the battery life is (6 Whr)/(0.048 W) = 125 hours (about 5 days). (b) If the phone is being used, the dynamic power is $P_{dynamic} = (0.5)(10^{-8}\ \text{F})(1.2\ \text{V})^2(3 \times 10^8\ \text{Hz}) = 2.16$ W. Together with the static and broadcast power, the total active power is 2.16 W + 0.048 W + 3 W = 5.2 W, so the battery life is 6 W-hr/5.2 W = 1.15 hours. This example somewhat oversimplifies the actual operation of a cell phone, but it illustrates the key ideas of power consumption.

1.9 SUMMARY AND A LOOK AHEAD

There are 10 kinds of people in this world: those who can count in binary and those who can't.

This chapter has introduced principles for understanding and designing complex systems. Although the real world is analog, digital designers discipline themselves to use a discrete subset of possible signals. In particular, binary variables have just two states: 0 and 1, also called FALSE and TRUE or LOW and HIGH. Logic gates compute a binary output from one or more binary inputs. Some of the common logic gates are:

- **NOT:** TRUE when input is FALSE
- **AND:** TRUE when all inputs are TRUE
- **OR:** TRUE when any inputs are TRUE
- **XOR:** TRUE when an odd number of inputs are TRUE

Logic gates are commonly built from CMOS transistors, which behave as electrically controlled switches. nMOS transistors turn ON when the gate is 1. pMOS transistors turn ON when the gate is 0.

In Chapters 2 through 5, we continue the study of digital logic. Chapter 2 addresses *combinational logic*, in which the outputs depend only on the current inputs. The logic gates introduced already are examples of combinational logic. You will learn to design circuits involving multiple gates to implement a relationship between inputs and outputs specified by a truth table or Boolean equation. Chapter 3 addresses *sequential logic*, in which the outputs depend on both current and past inputs. *Registers* are common sequential elements that remember their previous input. *Finite state machines*, built from registers and combinational logic, are a powerful way to build complicated systems in a systematic fashion. We also study timing of digital systems to analyze how fast a system can operate. Chapter 4 describes hardware description languages (HDLs). HDLs are related to conventional programming languages but are used to simulate and build hardware rather than software. Most digital systems today are designed with HDLs. SystemVerilog and VHDL are the two prevalent languages, and they are covered side-by-side in this book. Chapter 5 studies other combinational and sequential building blocks such as adders, multipliers, and memories.

Chapter 6 shifts to computer architecture. It describes the MIPS processor, an industry-standard microprocessor used in consumer electronics, some Silicon Graphics workstations, and many communications systems such as televisions, networking hardware, and wireless links. The MIPS architecture is defined by its registers and assembly language instruction set. You will learn to write programs in assembly language for the MIPS processor so that you can communicate with the processor in its native language.

Chapters 7 and 8 bridge the gap between digital logic and computer architecture. Chapter 7 investigates microarchitecture, the arrangement of digital building blocks, such as adders and registers, needed to construct a processor. In that chapter, you learn to build your own MIPS processor. Indeed, you learn three microarchitectures illustrating different trade-offs of performance and cost. Processor performance has increased exponentially, requiring ever more sophisticated memory systems to feed the insatiable demand for data. Chapter 8 delves into memory system architecture and also describes how computers communicate with peripheral devices such as keyboards and printers.

Exercises

Exercise 1.1 Explain in one paragraph at least three levels of abstraction that are used by

(a) biologists studying the operation of cells.

(b) chemists studying the composition of matter.

Exercise 1.2 Explain in one paragraph how the techniques of hierarchy, modularity, and regularity may be used by

(a) automobile designers.

(b) businesses to manage their operations.

Exercise 1.3 Ben Bitdiddle is building a house. Explain how he can use the principles of hierarchy, modularity, and regularity to save time and money during construction.

Exercise 1.4 An analog voltage is in the range of 0–5 V. If it can be measured with an accuracy of ±50 mV, at most how many bits of information does it convey?

Exercise 1.5 A classroom has an old clock on the wall whose minute hand broke off.

(a) If you can read the hour hand to the nearest 15 minutes, how many bits of information does the clock convey about the time?

(b) If you know whether it is before or after noon, how many additional bits of information do you know about the time?

Exercise 1.6 The Babylonians developed the *sexagesimal* (base 60) number system about 4000 years ago. How many bits of information is conveyed with one sexagesimal digit? How do you write the number 4000_{10} in sexagesimal?

Exercise 1.7 How many different numbers can be represented with 16 bits?

Exercise 1.8 What is the largest unsigned 32-bit binary number?

Exercise 1.9 What is the largest 16-bit binary number that can be represented with

(a) unsigned numbers?

(b) two's complement numbers?

(c) sign/magnitude numbers?

Exercise 1.10 What is the largest 32-bit binary number that can be represented with

(a) unsigned numbers?

(b) two's complement numbers?

(c) sign/magnitude numbers?

Exercise 1.11 What is the smallest (most negative) 16-bit binary number that can be represented with

(a) unsigned numbers?

(b) two's complement numbers?

(c) sign/magnitude numbers?

Exercise 1.12 What is the smallest (most negative) 32-bit binary number that can be represented with

(a) unsigned numbers?

(b) two's complement numbers?

(c) sign/magnitude numbers?

Exercise 1.13 Convert the following unsigned binary numbers to decimal. Show your work.

(a) 1010_2

(b) 110110_2

(c) 11110000_2

(d) 0001000101001111_2

Exercise 1.14 Convert the following unsigned binary numbers to decimal. Show your work.

(a) 1110_2

(b) 100100_2

(c) 11010111_2

(d) 0111010101001000_2

Exercise 1.15 Repeat Exercise 1.13, but convert to hexadecimal.

Exercise 1.16 Repeat Exercise 1.14, but convert to hexadecimal.

Exercise 1.17 Convert the following hexadecimal numbers to decimal. Show your work.

(a) $A5_{16}$

(b) $3B_{16}$

(c) $FFFF_{16}$

(d) $D0000000_{16}$

Exercise 1.18 Convert the following hexadecimal numbers to decimal. Show your work.

(a) $4E_{16}$

(b) $7C_{16}$

(c) $ED3A_{16}$

(d) $403FB001_{16}$

Exercise 1.19 Repeat Exercise 1.17, but convert to unsigned binary.

Exercise 1.20 Repeat Exercise 1.18, but convert to unsigned binary.

Exercise 1.21 Convert the following two's complement binary numbers to decimal.

(a) 1010_2

(b) 110110_2

(c) 01110000_2

(d) 10011111_2

Exercise 1.22 Convert the following two's complement binary numbers to decimal.

(a) 1110_2

(b) 100011_2

(c) 01001110_2

(d) 10110101_2

Exercise 1.23 Repeat Exercise 1.21, assuming the binary numbers are in sign/magnitude form rather than two's complement representation.

Exercise 1.24 Repeat Exercise 1.22, assuming the binary numbers are in sign/magnitude form rather than two's complement representation.

Exercise 1.25 Convert the following decimal numbers to unsigned binary numbers.

(a) 42_{10}

(b) 63_{10}

(c) 229_{10}

(d) 845_{10}

Exercise 1.26 Convert the following decimal numbers to unsigned binary numbers.

(a) 14_{10}

(b) 52_{10}

(c) 339_{10}

(d) 711_{10}

Exercise 1.27 Repeat Exercise 1.25, but convert to hexadecimal.

Exercise 1.28 Repeat Exercise 1.26, but convert to hexadecimal.

Exercise 1.29 Convert the following decimal numbers to 8-bit two's complement numbers or indicate that the decimal number would overflow the range.

(a) 42_{10}

(b) -63_{10}

(c) 124_{10}

(d) -128_{10}

(e) 133_{10}

Exercise 1.30 Convert the following decimal numbers to 8-bit two's complement numbers or indicate that the decimal number would overflow the range.

(a) 24_{10}

(b) -59_{10}

(c) 128_{10}

(d) -150_{10}

(e) 127_{10}

Exercise 1.31 Repeat Exercise 1.29, but convert to 8-bit sign/magnitude numbers.

Exercise 1.32 Repeat Exercise 1.30, but convert to 8-bit sign/magnitude numbers.

Exercise 1.33 Convert the following 4-bit two's complement numbers to 8-bit two's complement numbers.

(a) 0101_2

(b) 1010_2

Exercise 1.34 Convert the following 4-bit two's complement numbers to 8-bit two's complement numbers.

(a) 0111_2

(b) 1001_2

Exercise 1.35 Repeat Exercise 1.33 if the numbers are unsigned rather than two's complement.

Exercise 1.36 Repeat Exercise 1.34 if the numbers are unsigned rather than two's complement.

Exercise 1.37 Base 8 is referred to as *octal*. Convert each of the numbers from Exercise 1.25 to octal.

Exercise 1.38 Base 8 is referred to as *octal*. Convert each of the numbers from Exercise 1.26 to octal.

Exercise 1.39 Convert each of the following octal numbers to binary, hexadecimal, and decimal.

(a) 42_8

(b) 63_8

(c) 255_8

(d) 3047_8

Exercise 1.40 Convert each of the following octal numbers to binary, hexadecimal, and decimal.

(a) 23_8

(b) 45_8

(c) 371_8

(d) 2560_8

Exercise 1.41 How many 5-bit two's complement numbers are greater than 0? How many are less than 0? How would your answers differ for sign/magnitude numbers?

Exercise 1.42 How many 7-bit two's complement numbers are greater than 0? How many are less than 0? How would your answers differ for sign/magnitude numbers?

Exercise 1.43 How many bytes are in a 32-bit word? How many nibbles are in the word?

Exercise 1.44 How many bytes are in a 64-bit word?

Exercise 1.45 A particular DSL modem operates at 768 kbits/sec. How many bytes can it receive in 1 minute?

Exercise 1.46 USB 3.0 can send data at 5 Gbits/sec. How many bytes can it send in 1 minute?

Exercise 1.47 Hard disk manufacturers use the term "megabyte" to mean 10^6 bytes and "gigabyte" to mean 10^9 bytes. How many real GBs of music can you store on a 50 GB hard disk?

Exercise 1.48 Estimate the value of 2^{31} without using a calculator.

Exercise 1.49 A memory on the Pentium II microprocessor is organized as a rectangular array of bits with 2^8 rows and 2^9 columns. Estimate how many bits it has without using a calculator.

Exercise 1.50 Draw a number line analogous to Figure 1.11 for 3-bit unsigned, two's complement, and sign/magnitude numbers.

Exercise 1.51 Draw a number line analogous to Figure 1.11 for 2-bit unsigned, two's complement, and sign/magnitude numbers.

Exercise 1.52 Perform the following additions of unsigned binary numbers. Indicate whether or not the sum overflows a 4-bit result.

(a) $1001_2 + 0100_2$

(b) $1101_2 + 1011^2$

Exercise 1.53 Perform the following additions of unsigned binary numbers. Indicate whether or not the sum overflows an 8-bit result.

(a) $10011001_2 + 01000100_2$

(b) $11010010_2 + 10110110_2$

Exercise 1.54 Repeat Exercise 1.52, assuming that the binary numbers are in two's complement form.

Exercise 1.55 Repeat Exercise 1.53, assuming that the binary numbers are in two's complement form.

Exercise 1.56 Convert the following decimal numbers to 6-bit two's complement binary numbers and add them. Indicate whether or not the sum overflows a 6-bit result.

(a) $16_{10} + 9_{10}$

(b) $27_{10} + 31_{10}$

(c) $-4_{10} + 19_{10}$

(d) $3_{10} + -32_{10}$

(e) $-16_{10} + -9_{10}$

(f) $-27_{10} + -31_{10}$

Exercise 1.57 Repeat Exercise 1.56 for the following numbers.

(a) $7_{10} + 13_{10}$

(b) $17_{10} + 25_{10}$

(c) $-26_{10} + 8_{10}$

(d) $31_{10} + -14_{10}$

(e) $-19_{10} + -22_{10}$

(f) $-2_{10} + -29_{10}$

Exercise 1.58 Perform the following additions of unsigned hexadecimal numbers. Indicate whether or not the sum overflows an 8-bit (two hex digit) result.

(a) $7_{16} + 9_{16}$

(b) $13_{16} + 28_{16}$

(c) $AB_{16} + 3E_{16}$

(d) $8F_{16} + AD_{16}$

Exercise 1.59 Perform the following additions of unsigned hexadecimal numbers. Indicate whether or not the sum overflows an 8-bit (two hex digit) result.

(a) $22_{16} + 8_{16}$

(b) $73_{16} + 2C_{16}$

(c) $7F_{16} + 7F_{16}$

(d) $C2_{16} + A4_{16}$

Exercise 1.60 Convert the following decimal numbers to 5-bit two's complement binary numbers and subtract them. Indicate whether or not the difference overflows a 5-bit result.

(a) $9_{10} - 7_{10}$

(b) $12_{10} - 15_{10}$

(c) $-6_{10} - 11_{10}$

(d) $4_{10} - -8_{10}$

Exercise 1.61 Convert the following decimal numbers to 6-bit two's complement binary numbers and subtract them. Indicate whether or not the difference overflows a 6-bit result.

(a) $18_{10} - 12_{10}$

(b) $30_{10} - 9_{10}$

(c) $-28_{10} - 3_{10}$

(d) $-16_{10} - 21_{10}$

Exercise 1.62 In a *biased* N-bit binary number system with bias B, positive and negative numbers are represented as their value plus the bias B. For example, for 5-bit numbers with a bias of 15, the number 0 is represented as 01111, 1 as 10000, and so forth. Biased number systems are sometimes used in floating point mathematics, which will be discussed in Chapter 5. Consider a biased 8-bit binary number system with a bias of 127_{10}.

(a) What decimal value does the binary number 10000010_2 represent?

(b) What binary number represents the value 0?

(c) What is the representation and value of the most negative number?

(d) What is the representation and value of the most positive number?

Exercise 1.63 Draw a number line analogous to Figure 1.11 for 3-bit biased numbers with a bias of 3 (see Exercise 1.62 for a definition of biased numbers).

Exercise 1.64 In a *binary coded decimal* (BCD) system, 4 bits are used to represent a decimal digit from 0 to 9. For example, 37_{10} is written as 00110111_{BCD}.

(a) Write 289_{10} in BCD

(b) Convert 100101010001_{BCD} to decimal

(c) Convert 01101001_{BCD} to binary

(d) Explain why BCD might be a useful way to represent numbers

Exercise 1.65 Answer the following questions related to BCD systems (see Exercise 1.64 for the definition of BCD).

(a) Write 371_{10} in BCD

(b) Convert 000110000111_{BCD} to decimal

(c) Convert 10010101_{BCD} to binary

(d) Explain the disadvantages of BCD when compared to binary representations of numbers

Exercise 1.66 A flying saucer crashes in a Nebraska cornfield. The FBI investigates the wreckage and finds an engineering manual containing an equation in the Martian number system: $325 + 42 = 411$. If this equation is correct, how many fingers would you expect Martians to have?

Exercise 1.67 Ben Bitdiddle and Alyssa P. Hacker are having an argument. Ben says, "All integers greater than zero and exactly divisible by six have exactly two 1's in their binary representation." Alyssa disagrees. She says, "No, but all such numbers have an even number of 1's in their representation." Do you agree with Ben or Alyssa or both or neither? Explain.

Exercise 1.68 Ben Bitdiddle and Alyssa P. Hacker are having another argument. Ben says, "I can get the two's complement of a number by subtracting 1, then inverting all the bits of the result." Alyssa says, "No, I can do it by examining each bit of the number, starting with the least significant bit. When the first 1 is found, invert each subsequent bit." Do you agree with Ben or Alyssa or both or neither? Explain.

Exercise 1.69 Write a program in your favorite language (e.g., C, Java, Perl) to convert numbers from binary to decimal. The user should type in an unsigned binary number. The program should print the decimal equivalent.

Exercise 1.70 Repeat Exercise 1.69 but convert from an arbitrary base b_1 to another base b_2, as specified by the user. Support bases up to 16, using the letters of the alphabet for digits greater than 9. The user should enter b_1, b_2, and then the number to convert in base b_1. The program should print the equivalent number in base $b_{2.}$

Exercise 1.71 Draw the symbol, Boolean equation, and truth table for

(a) a three-input OR gate

(b) a three-input exclusive OR (XOR) gate

(c) a four-input XNOR gate

Exercise 1.72 Draw the symbol, Boolean equation, and truth table for

(a) a four-input OR gate

(b) a three-input XNOR gate

(c) a five-input NAND gate

Exercise 1.73 A *majority gate* produces a TRUE output if and only if more than half of its inputs are TRUE. Complete a truth table for the three-input majority gate shown in Figure 1.41.

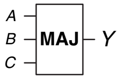


Figure 1.41 Three-input majority gate

Exercise 1.74 A three-input *AND-OR* (*AO*) *gate* shown in Figure 1.42 produces a TRUE output if both *A* and *B* are TRUE, or if *C* is TRUE. Complete a truth table for the gate.

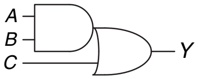


Figure 1.42 Three-input AND-OR gate

Exercise 1.75 A three-input *OR-AND-INVERT* (*OAI*) *gate* shown in Figure 1.43 produces a FALSE output if *C* is TRUE and *A* or *B* is TRUE. Otherwise it produces a TRUE output. Complete a truth table for the gate.

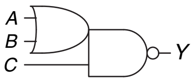


Figure 1.43 **Three-input OR-AND-INVERT gate**

Exercise 1.76 There are 16 different truth tables for Boolean functions of two variables. List each truth table. Give each one a short descriptive name (such as OR, NAND, and so on).

Exercise 1.77 How many different truth tables exist for Boolean functions of N variables?

Exercise 1.78 Is it possible to assign logic levels so that a device with the transfer characteristics shown in Figure 1.44 would serve as an inverter? If so, what are the input and output low and high levels (V_{IL}, V_{OL}, V_{IH}, and V_{OH}) and noise margins (NM_L and NM_H)? If not, explain why not.

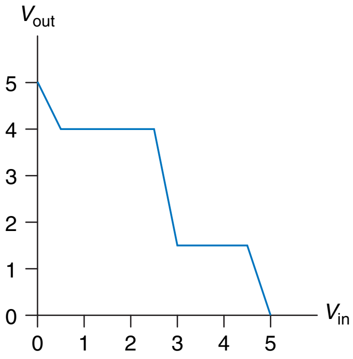


Figure 1.44 **DC transfer characteristics**

Exercise 1.79 Repeat Exercise 1.78 for the transfer characteristics shown in Figure 1.45.

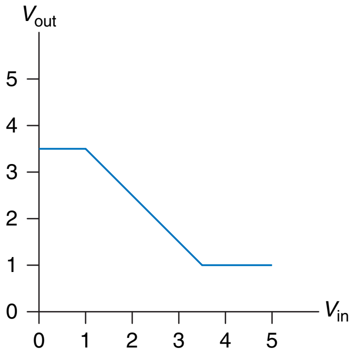


Figure 1.45 **DC transfer characteristics**

Exercise 1.80 Is it possible to assign logic levels so that a device with the transfer characteristics shown in Figure 1.46 would serve as a buffer? If so, what are the input and output low and high levels (V_{IL},V_{OL},V_{IH}, and V_{OH}) and noise margins (NM_L and NM_H)? If not, explain why not.

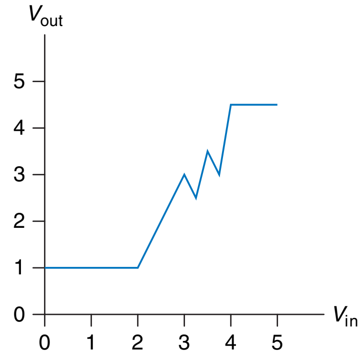


Figure 1.46 **DC transfer characteristics**

Exercise 1.81 Ben Bitdiddle has invented a circuit with the transfer characteristics shown in Figure 1.47 that he would like to use as a buffer. Will it work? Why or why not? He would like to advertise that it is compatible with LVCMOS and LVTTL logic. Can Ben's buffer correctly receive inputs from those logic families? Can its output properly drive those logic families? Explain.

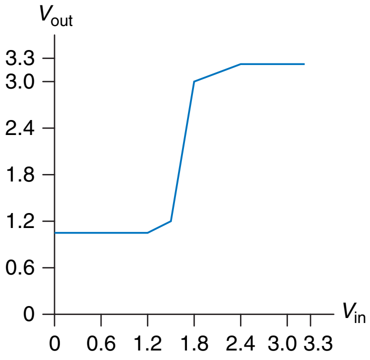


Figure 1.47 **Ben's buffer DC transfer characteristics**

Exercise 1.82 While walking down a dark alley, Ben Bitdiddle encounters a two-input gate with the transfer function shown in Figure 1.48. The inputs are A and B and the output is Y.

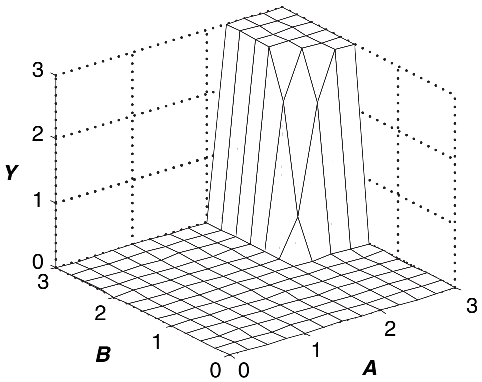


Figure 1.48 Two-input DC transfer characteristics

(a) What kind of logic gate did he find?

(b) What are the approximate high and low logic levels?

Exercise 1.83 Repeat Exercise 1.82 for Figure 1.49.

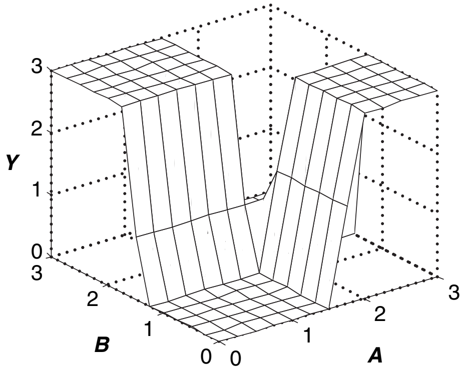


Figure 1.49 Two-input DC transfer characteristics

Exercise 1.84 Sketch a transistor-level circuit for the following CMOS gates. Use a minimum number of transistors.

(a) four-input NAND gate

(b) three-input OR-AND-INVERT gate (see Exercise 1.75)

(c) three-input AND-OR gate (see Exercise 1.74)

Exercise 1.85 Sketch a transistor-level circuit for the following CMOS gates. Use a minimum number of transistors.

(a) three-input NOR gate

(b) three-input AND gate

(c) two-input OR gate

Exercise 1.86 A *minority gate* produces a TRUE output if and only if fewer than half of its inputs are TRUE. Otherwise it produces a FALSE output. Sketch a transistor-level circuit for a three-input CMOS minority gate. Use a minimum number of transistors.

Exercise 1.87 Write a truth table for the function performed by the gate in Figure 1.50. The truth table should have two inputs, *A* and *B*. What is the name of this function?

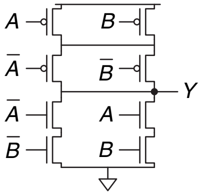


Figure 1.50 Mystery schematic

Exercise 1.88 Write a truth table for the function performed by the gate in Figure 1.51. The truth table should have three inputs, *A*, *B*, and C.

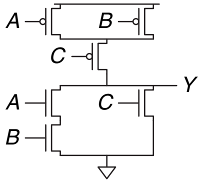


Figure 1.51 Mystery schematic

Exercise 1.89 Implement the following three-input gates using only pseudo-nMOS logic gates. Your gates receive three inputs, *A*, *B*, and C. Use a minimum number of transistors.

(a) three-input NOR gate

(b) three-input NAND gate

(c) three-input AND gate

Exercise 1.90 *Resistor-Transistor Logic* (*RTL*) uses nMOS transistors to pull the gate output LOW and a weak resistor to pull the output HIGH when none of the paths to ground are active. A NOT gate built using RTL is shown in Figure 1.52. Sketch a three-input RTL NOR gate. Use a minimum number of transistors.

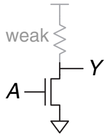


Figure 1.52 RTL NOT gate

Interview Questions

These questions have been asked at interviews for digital design jobs.

Question 1.1 Sketch a transistor-level circuit for a CMOS four-input NOR gate.

Question 1.2 The king receives 64 gold coins in taxes but has reason to believe that one is counterfeit. He summons you to identify the fake coin. You have a balance that can hold coins on each side. How many times do you need to use the balance to find the lighter, fake coin?

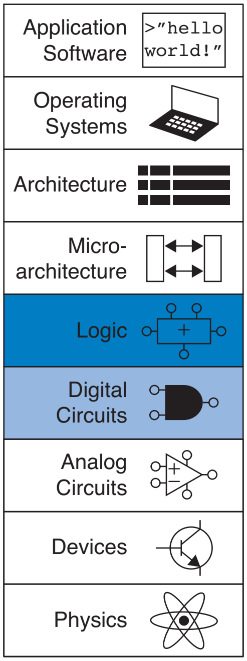
Question 1.3 The professor, the teaching assistant, the digital design student, and the freshman track star need to cross a rickety bridge on a dark night. The bridge is so shaky that only two people can cross at a time. They have only one flashlight among them and the span is too long to throw the flashlight, so somebody must carry it back to the other people. The freshman track star can cross the bridge in 1 minute. The digital design student can cross the bridge in 2 minutes. The teaching assistant can cross the bridge in 5 minutes. The professor always gets distracted and takes 10 minutes to cross the bridge. What is the fastest time to get everyone across the bridge?

This page intentionally left blank

Combinational Logic Design

2

2.1 Introduction
2.2 Boolean Equations
2.3 Boolean Algebra
2.4 From Logic to Gates
2.5 Multilevel Combinational Logic
2.6 X's and Z's, Oh My
2.7 Karnaugh Maps
2.8 Combinational Building Blocks
2.9 Timing
2.10 Summary
Exercises
Interview Questions


Application Software
>"hello world!"
Operating Systems
Architecture
Micro-architecture
Logic
+
Digital Circuits
Analog Circuits
+
−
Devices
Physics

2.1 INTRODUCTION

In digital electronics, a *circuit* is a network that processes discrete-valued variables. A circuit can be viewed as a black box, shown in Figure 2.1, with

- one or more discrete-valued *input terminals*
- one or more discrete-valued *output terminals*
- a *functional specification* describing the relationship between inputs and outputs
- a *timing specification* describing the delay between inputs changing and outputs responding.

Peering inside the black box, circuits are composed of nodes and elements. An *element* is itself a circuit with inputs, outputs, and a specification. A *node* is a wire, whose voltage conveys a discrete-valued variable. Nodes are classified as *input, output*, or *internal*. Inputs receive values from the external world. Outputs deliver values to the external world. Wires that are not inputs or outputs are called internal nodes. Figure 2.2

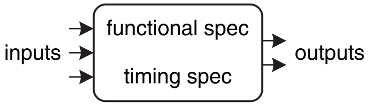

inputs
functional spec
timing spec
outputs

Figure 2.1 **Circuit as a black box with inputs, outputs, and specifications**

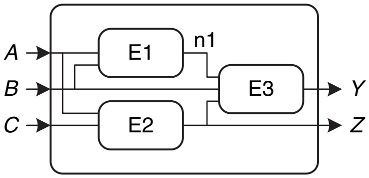
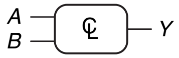

A
B
C
E1
n1
E2
E3
Y
Z

Figure 2.2 **Elements and nodes**

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00002-1
© 2013 Elsevier, Inc. All rights reserved.

illustrates a circuit with three elements, E1, E2, and E3, and six nodes. Nodes A, B, and C are inputs. Y and Z are outputs. n1 is an internal node between E1 and E3.

Digital circuits are classified as *combinational* or *sequential*. A combinational circuit's outputs depend only on the current values of the inputs; in other words, it combines the current input values to compute the output. For example, a logic gate is a combinational circuit. A sequential circuit's outputs depend on both current and previous values of the inputs; in other words, it depends on the input sequence. A combinational circuit is *memoryless*, but a sequential circuit has *memory*. This chapter focuses on combinational circuits, and Chapter 3 examines sequential circuits.



$Y = F(A, B) = A + B$

Figure 2.3 **Combinational logic circuit**

The functional specification of a combinational circuit expresses the output values in terms of the current input values. The timing specification of a combinational circuit consists of lower and upper bounds on the delay from input to output. We will initially concentrate on the functional specification, then return to the timing specification later in this chapter.

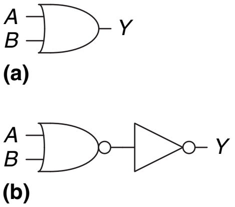
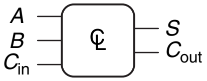


Figure 2.4 **Two OR implementations**

Figure 2.3 shows a combinational circuit with two inputs and one output. On the left of the figure are the inputs, A and B, and on the right is the output, Y. The symbol CL inside the box indicates that it is implemented using only combinational logic. In this example, the function F is specified to be OR: $Y = F(A, B) = A + B$. In words, we say the output Y is a function of the two inputs, A and B, namely $Y = A$ OR B.

Figure 2.4 shows two possible *implementations* for the combinational logic circuit in Figure 2.3. As we will see repeatedly throughout the book, there are often many implementations for a single function. You choose which to use given the building blocks at your disposal and your design constraints. These constraints often include area, speed, power, and design time.



$S = A \oplus B \oplus C_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

Figure 2.5 **Multiple-output combinational circuit**

Figure 2.5 shows a combinational circuit with multiple outputs. This particular combinational circuit is called a *full adder* and we will revisit it in Section 5.2.1. The two equations specify the function of the outputs, S and C_{out}, in terms of the inputs, A, B, and C_{in}.

To simplify drawings, we often use a single line with a slash through it and a number next to it to indicate a *bus*, a bundle of multiple signals. The number specifies how many signals are in the bus. For example, Figure 2.6(a) represents a block of combinational logic with three inputs and two outputs. If the number of bits is unimportant or obvious from the context, the slash may be shown without a number. Figure 2.6(b) indicates two blocks of combinational logic with an arbitrary number of outputs from one block serving as inputs to the second block.

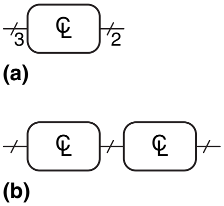


Figure 2.6 **Slash notation for multiple signals**

The rules of *combinational composition* tell us how we can build a large combinational circuit from smaller combinational circuit elements.

A circuit is combinational if it consists of interconnected circuit elements such that

- Every circuit element is itself combinational.
- Every node of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element.
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once.

The rules of combinational composition are sufficient but not strictly necessary. Certain circuits that disobey these rules are still combinational, so long as the outputs depend only on the current values of the inputs. However, determining whether oddball circuits are combinational is more difficult, so we will usually restrict ourselves to combinational composition as a way to build combinational circuits.

Example 2.1 COMBINATIONAL CIRCUITS

Which of the circuits in Figure 2.7 are combinational circuits according to the rules of combinational composition?

Solution: Circuit (a) is combinational. It is constructed from two combinational circuit elements (inverters I1 and I2). It has three nodes: n1, n2, and n3. n1 is an input to the circuit and to I1; n2 is an internal node, which is the output of I1 and the input to I2; n3 is the output of the circuit and of I2. (b) is not combinational, because there is a cyclic path: the output of the XOR feeds back to one of its inputs. Hence, a cyclic path starting at n4 passes through the XOR to n5, which returns to n4. (c) is combinational. (d) is not combinational, because node n6 connects to the output terminals of both I3 and I4. (e) is combinational, illustrating two combinational circuits connected to form a larger combinational circuit. (f) does not obey the rules of combinational composition because it has a cyclic path through the two elements. Depending on the functions of the elements, it may or may not be a combinational circuit.

Large circuits such as microprocessors can be very complicated, so we use the principles from Chapter 1 to manage the complexity. Viewing a circuit as a black box with a well-defined interface and function is an application of abstraction and modularity. Building the circuit out of

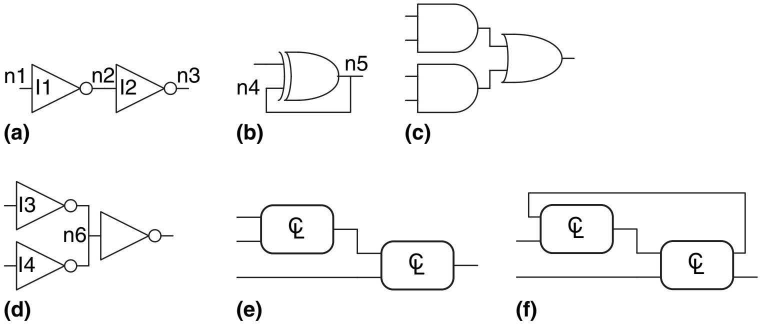


Figure 2.7 **Example circuits**

smaller circuit elements is an application of hierarchy. The rules of combinational composition are an application of discipline.

The functional specification of a combinational circuit is usually expressed as a truth table or a Boolean equation. In the next sections, we describe how to derive a Boolean equation from any truth table and how to use Boolean algebra and Karnaugh maps to simplify equations. We show how to implement these equations using logic gates and how to analyze the speed of these circuits.

2.2 BOOLEAN EQUATIONS

Boolean equations deal with variables that are either TRUE or FALSE, so they are perfect for describing digital logic. This section defines some terminology commonly used in Boolean equations, then shows how to write a Boolean equation for any logic function given its truth table.

2.2.1 Terminology

The *complement* of a variable A is its inverse $\overline{A}$. The variable or its complement is called a *literal*. For example, A, $\overline{A}$, B, and $\overline{B}$ are literals. We call A the *true form* of the variable and $\overline{A}$ the *complementary form;* "true form" does not mean that A is TRUE, but merely that A does not have a line over it.

The AND of one or more literals is called a *product* or an *implicant*. $\overline{A}B$, $A\overline{B}\,\overline{C}$, and B are all implicants for a function of three variables. A *minterm* is a product involving all of the inputs to the function. $A\overline{B}\,\overline{C}$ is a minterm for a function of the three variables A, B, and C, but $\overline{A}B$ is not, because it does not involve C. Similarly, the OR of one or more literals is called a *sum*. A *maxterm* is a sum involving all of the inputs to the function. $A + \overline{B} + C$ is a maxterm for a function of the three variables A, B, and C.

The *order of operations* is important when interpreting Boolean equations. Does $Y = A + BC$ mean $Y = (A$ OR $B)$ AND C or $Y = A$ OR $(B$ AND $C)$? In Boolean equations, NOT has the highest *precedence*, followed by AND, then OR. Just as in ordinary equations, products are performed before sums. Therefore, the equation is read as $Y = A$ OR $(B$ AND $C)$. Equation 2.1 gives another example of order of operations.

$$\overline{A}B + BC\overline{D} = ((\overline{A})B) + (BC(\overline{D})) \tag{2.1}$$

A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}$ B	m_1
1	0	0	A $\overline{B}$	m_2
1	1	0	A B	m_3

Figure 2.8 Truth table and minterms

2.2.2 Sum-of-Products Form

A truth table of N inputs contains 2^N rows, one for each possible value of the inputs. Each row in a truth table is associated with a minterm that is TRUE for that row. Figure 2.8 shows a truth table of two inputs, A and B. Each row shows its corresponding minterm. For example, the minterm for the first row is $\overline{A}\,\overline{B}$ because $\overline{A}\,\overline{B}$ is TRUE when $A = 0$, $B = 0$. The minterms are numbered starting with 0; the top row corresponds to minterm 0, m_0, the next row to minterm 1, m_1, and so on.

We can write a Boolean equation for any truth table by summing each of the minterms for which the output, Y, is TRUE. For example, in Figure 2.8, there is only one row (or minterm) for which the output Y is TRUE, shown circled in blue. Thus, $Y = \overline{A}B$. Figure 2.9 shows a truth table with more than one row in which the output is TRUE. Taking the sum of each of the circled minterms gives $Y = \overline{A}B + AB$.

A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\,\overline{B}$	m_0
0	1	1	$\overline{A}\,B$	m_1
1	0	0	$A\,\overline{B}$	m_2
1	1	1	$A\,B$	m_3

Figure 2.9 **Truth table with multiple TRUE minterms**

Canonical form is just a fancy word for standard form. You can use the term to impress your friends and scare your enemies.

This is called the *sum-of-products canonical form* of a function because it is the sum (OR) of products (ANDs forming minterms). Although there are many ways to write the same function, such as $Y = B\overline{A} + BA$, we will sort the minterms in the same order that they appear in the truth table, so that we always write the same Boolean expression for the same truth table.

The sum-of-products canonical form can also be written in *sigma notation* using the summation symbol, Σ. With this notation, the function from Figure 2.9 would be written as:

$$F(A,B) = \Sigma(m_1, m_3)$$

or (2.2)

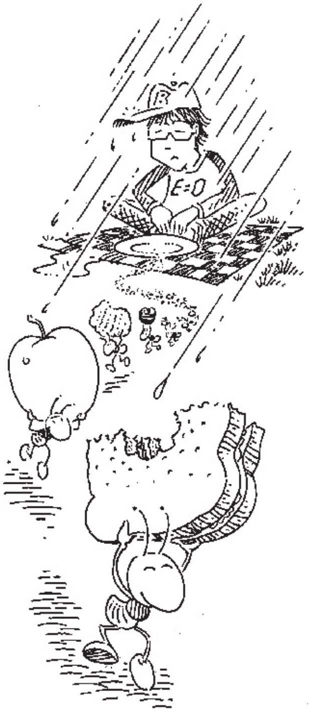
$$F(A,B) = \Sigma(1,3)$$

Example 2.2 SUM-OF-PRODUCTS FORM

Ben Bitdiddle is having a picnic. He won't enjoy it if it rains or if there are ants. Design a circuit that will output TRUE *only* if Ben enjoys the picnic.

Solution: First define the inputs and outputs. The inputs are A and R, which indicate if there are ants and if it rains. A is TRUE when there are ants and FALSE when there are no ants. Likewise, R is TRUE when it rains and FALSE when the sun smiles on Ben. The output is E, Ben's enjoyment of the picnic. E is TRUE if Ben enjoys the picnic and FALSE if he suffers. Figure 2.10 shows the truth table for Ben's picnic experience.

Using sum-of-products form, we write the equation as: $E = \overline{A}\,\overline{R}$ or $E = \Sigma(0)$. We can build the equation using two inverters and a two-input AND gate, shown in Figure 2.11(a). You may recognize this truth table as the NOR function from Section 1.5.5: $E = A$ NOR R $= \overline{A+R}$. Figure 2.11(b) shows the NOR implementation. In Section 2.3, we show that the two equations, $\overline{A}\,\overline{R}$ and $\overline{A+R}$, are equivalent.



A	R	E
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.10 **Ben's truth table**

The sum-of-products form provides a Boolean equation for any truth table with any number of variables. Figure 2.12 shows a random three-input truth table. The sum-of-products form of the logic function is

$$Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$$

or (2.3)

$$Y = \Sigma(0,4,5)$$

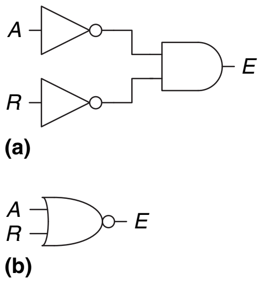


Figure 2.11 Ben's circuit

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Figure 2.12 Random three-input truth table

Unfortunately, sum-of-products form does not necessarily generate the simplest equation. In Section 2.3 we show how to write the same function using fewer terms.

2.2.3 Product-of-Sums Form

An alternative way of expressing Boolean functions is the *product-of-sums canonical form*. Each row of a truth table corresponds to a maxterm that is FALSE for that row. For example, the maxterm for the first row of a two-input truth table is $(A + B)$ because $(A + B)$ is FALSE when $A = 0$, $B = 0$. We can write a Boolean equation for any circuit directly from the truth table as the AND of each of the maxterms for which the output is FALSE. The product-of-sums canonical form can also be written in *pi notation* using the product symbol, Π.

Example 2.3 PRODUCT-OF-SUMS FORM

Write an equation in product-of-sums form for the truth table in Figure 2.13.

Solution: The truth table has two rows in which the output is FALSE. Hence, the function can be written in product-of-sums form as $Y = (A + B)(\overline{A} + B)$ or, using pi notation, $Y = \Pi(M_0, M_2)$ or $Y = \Pi(0, 2)$. The first maxterm, $(A + B)$, guarantees that $Y = 0$ for $A = 0$, $B = 0$, because any value AND 0 is 0. Likewise, the second maxterm, $(\overline{A} + \mathrm{B})$, guarantees that $Y = 0$ for $A = 1$, $B = 0$. Figure 2.13 is the same truth table as Figure 2.9, showing that the same function can be written in more than one way.

A	B	Y	maxterm	maxterm name
0	0	0	$A + B$	M_0
0	1	1	$A + \overline{B}$	M_1
1	0	0	$\overline{A} + B$	M_2
1	1	1	$\overline{A} + \overline{B}$	M_3

Figure 2.13 Truth table with multiple FALSE maxterms

Similarly, a Boolean equation for Ben's picnic from Figure 2.10 can be written in product-of-sums form by circling the three rows of 0's to obtain $E = (A + \overline{R})(\overline{A} + R)(\overline{A} + \overline{R})$ or $E = \Pi(1, 2, 3)$. This is uglier than the sum-of-products equation, $E = \overline{A}\,\overline{R}$, but the two equations are logically equivalent.

Sum-of-products produces a shorter equation when the output is TRUE on only a few rows of a truth table; product-of-sums is simpler when the output is FALSE on only a few rows of a truth table.

2.3 BOOLEAN ALGEBRA

In the previous section, we learned how to write a Boolean expression given a truth table. However, that expression does not necessarily lead to the simplest set of logic gates. Just as you use algebra to simplify mathematical equations, you can use *Boolean algebra* to simplify Boolean equations. The rules of Boolean algebra are much like those of ordinary algebra but are in some cases simpler, because variables have only two possible values: 0 or 1.

Boolean algebra is based on a set of axioms that we assume are correct. Axioms are unprovable in the sense that a definition cannot be proved. From these axioms, we prove all the theorems of Boolean algebra.

Table 2.1 **Axioms of Boolean algebra**

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

These theorems have great practical significance, because they teach us how to simplify logic to produce smaller and less costly circuits.

Axioms and theorems of Boolean algebra obey the principle of *duality*. If the symbols 0 and 1 and the operators • (AND) and + (OR) are interchanged, the statement will still be correct. We use the prime symbol (′) to denote the *dual* of a statement.

2.3.1 Axioms

Table 2.1 states the axioms of Boolean algebra. These five axioms and their duals define Boolean variables and the meanings of NOT, AND, and OR. Axiom A1 states that a Boolean variable *B* is 0 if it is not 1. The axiom's dual, A1′, states that the variable is 1 if it is not 0. Together, A1 and A1′ tell us that we are working in a Boolean or binary field of 0's and 1's. Axioms A2 and A2′ define the NOT operation. Axioms A3 to A5 define AND; their duals, A3′ to A5′ define OR.

2.3.2 Theorems of One Variable

Theorems T1 to T5 in Table 2.2 describe how to simplify equations involving one variable.

The *identity* theorem, T1, states that for any Boolean variable *B*, *B* AND 1 = *B*. Its dual states that *B* OR 0 = *B*. In hardware, as shown in Figure 2.14, T1 means that if one input of a two-input AND gate is always 1, we can remove the AND gate and replace it with a wire connected to the variable input (*B*). Likewise, T1′ means that if one input of a two-input OR gate is always 0, we can replace the OR gate with a wire connected to *B*. In general, gates cost money, power, and delay, so replacing a gate with a wire is beneficial.

The *null element theorem*, T2, says that *B* AND 0 is always equal to 0. Therefore, 0 is called the *null* element for the AND operation, because it nullifies the effect of any other input. The dual states that *B* OR 1 is always equal to 1. Hence, 1 is the null element for the OR operation. In hardware,

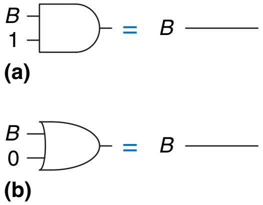
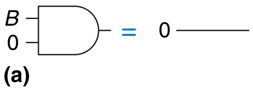


Figure 2.14 **Identity theorem in hardware: (a) T1, (b) T1′**

The null element theorem leads to some outlandish statements that are actually true! It is particularly dangerous when left in the hands of advertisers: YOU WILL GET A MILLION DOLLARS or we'll send you a toothbrush in the mail. (You'll most likely be receiving a toothbrush in the mail.)



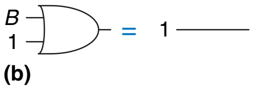


Figure 2.15 Null element theorem in hardware: (a) T2, (b) T2′

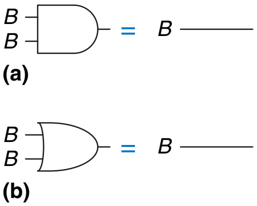


Figure 2.16 Idempotency theorem in hardware: (a) T3, (b) T3′

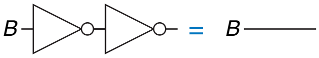


Figure 2.17 Involution theorem in hardware: T4

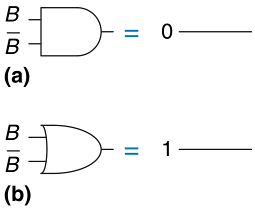


Figure 2.18 Complement theorem in hardware: (a) T5, (b) T5′

Table 2.2 Boolean theorems of one variable

	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\bar{\bar{B}} = B$		Involution
T5	$B \bullet \bar{B} = 0$	T5′	$B + \bar{B} = 1$	Complements

as shown in Figure 2.15, if one input of an AND gate is 0, we can replace the AND gate with a wire that is tied LOW (to 0). Likewise, if one input of an OR gate is 1, we can replace the OR gate with a wire that is tied HIGH (to 1).

Idempotency, T3, says that a variable AND itself is equal to just itself. Likewise, a variable OR itself is equal to itself. The theorem gets its name from the Latin roots: *idem* (same) and *potent* (power). The operations return the same thing you put into them. Figure 2.16 shows that idempotency again permits replacing a gate with a wire.

Involution, T4, is a fancy way of saying that complementing a variable twice results in the original variable. In digital electronics, two wrongs make a right. Two inverters in series logically cancel each other out and are logically equivalent to a wire, as shown in Figure 2.17. The dual of T4 is itself.

The *complement theorem*, T5 (Figure 2.18), states that a variable AND its complement is 0 (because one of them has to be 0). And by duality, a variable OR its complement is 1 (because one of them has to be 1).

2.3.3 Theorems of Several Variables

Theorems T6 to T12 in Table 2.3 describe how to simplify equations involving more than one Boolean variable.

Commutativity and *associativity*, T6 and T7, work the same as in traditional algebra. By commutativity, the *order* of inputs for an AND or OR function does not affect the value of the output. By associativity, the specific groupings of inputs do not affect the value of the output.

The *distributivity theorem*, T8, is the same as in traditional algebra, but its dual, T8′, is not. By T8, AND distributes over OR, and by T8′, OR distributes over AND. In traditional algebra, multiplication distributes over addition but addition does not distribute over multiplication, so that $(B + C) \times (B + D) \neq B + (C \times D)$.

The *covering*, *combining*, and *consensus* theorems, T9 to T11, permit us to eliminate redundant variables. With some thought, you should be able to convince yourself that these theorems are correct.

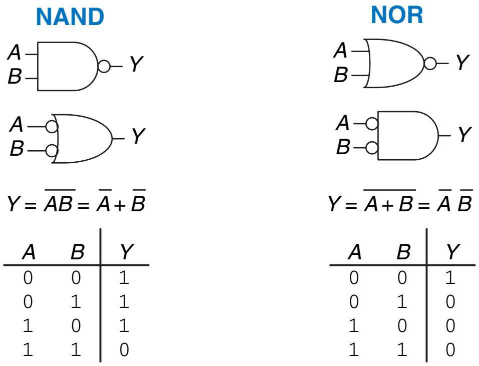
Table 2.3 **Boolean theorems of several variables**

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= B \bullet C + \overline{B} \bullet D$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} \ldots)$	De Morgan's Theorem

De Morgan's Theorem, T12, is a particularly powerful tool in digital design. The theorem explains that the complement of the product of all the terms is equal to the sum of the complement of each term. Likewise, the complement of the sum of all the terms is equal to the product of the complement of each term.

According to De Morgan's theorem, a NAND gate is equivalent to an OR gate with inverted inputs. Similarly, a NOR gate is equivalent to an AND gate with inverted inputs. Figure 2.19 shows these *De Morgan equivalent gates* for NAND and NOR gates. The two symbols shown for each function are called *duals*. They are logically equivalent and can be used interchangeably.

The inversion circle is called a *bubble*. Intuitively, you can imagine that "pushing" a bubble through the gate causes it to come out at the other side



$Y = \overline{AB} = \overline{A} + \overline{B}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

$Y = \overline{A + B} = \overline{A}\;\overline{B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.19 **De Morgan equivalent gates**

Augustus De Morgan, died 1871. A British mathematician, born in India. Blind in one eye. His father died when he was 10. Attended Trinity College, Cambridge, at age 16, and was appointed Professor of Mathematics at the newly founded London University at age 22. Wrote widely on many mathematical subjects, including logic, algebra, and paradoxes. De Morgan's crater on the moon is named for him. He proposed a riddle for the year of his birth: "I was x years of age in the year x^2."

and flips the body of the gate from AND to OR or vice versa. For example, the NAND gate in Figure 2.19 consists of an AND body with a bubble on the output. Pushing the bubble to the left results in an OR body with bubbles on the inputs. The underlying rules for bubble pushing are

- Pushing bubbles backward (from the output) or forward (from the inputs) changes the body of the gate from AND to OR or vice versa.
- Pushing a bubble from the output back to the inputs puts bubbles on all gate inputs.
- Pushing bubbles on *all* gate inputs forward toward the output puts a bubble on the output.

Section 2.5.2 uses bubble pushing to help analyze circuits.

Example 2.4 DERIVE THE PRODUCT-OF-SUMS FORM

Figure 2.20 shows the truth table for a Boolean function Y and its complement $\overline{Y}$. Using De Morgan's Theorem, derive the product-of-sums canonical form of Y from the sum-of-products form of $\overline{Y}$.

A	B	Y	$\overline{Y}$
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	0

Figure 2.20 **Truth table showing Y and $\overline{Y}$**

Solution: Figure 2.21 shows the minterms (circled) contained in $\overline{Y}$. The sum-of-products canonical form of $\overline{Y}$ is

A	B	Y	$\overline{Y}$	minterm
0	0	0	**1**	$\overline{A}\,\overline{B}$
0	1	0	**1**	$\overline{A}\,B$
1	0	1	**0**	$A\,\overline{B}$
1	1	1	**0**	$A\,B$

Figure 2.21 **Truth table showing minterms for $\overline{Y}$**

$$\overline{Y} = \overline{A}\,\overline{B} + \overline{A}B \tag{2.4}$$

Taking the complement of both sides and applying De Morgan's Theorem twice, we get:

$$\overline{\overline{Y}} = Y = \overline{\overline{A}\,\overline{B} + \overline{A}B} = (\overline{\overline{A}\,\overline{B}})(\overline{\overline{A}B}) = (A+B)(A+\overline{B}) \tag{2.5}$$

2.3.4 The Truth Behind It All

The curious reader might wonder how to prove that a theorem is true. In Boolean algebra, proofs of theorems with a finite number of variables are easy: just show that the theorem holds for all possible values of these variables. This method is called *perfect induction* and can be done with a truth table.

Example 2.5 PROVING THE CONSENSUS THEOREM USING PERFECT INDUCTION

Prove the consensus theorem, T11, from Table 2.3.

Solution: Check both sides of the equation for all eight combinations of B, C, and D. The truth table in Figure 2.22 illustrates these combinations. Because $BC + \overline{B}D + CD = BC + \overline{B}D$ for all cases, the theorem is proved.

B	C	D	$BC + \overline{B}D + CD$	$BC + \overline{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Figure 2.22 **Truth table proving T11**

2.3.5 Simplifying Equations

The theorems of Boolean algebra help us simplify Boolean equations. For example, consider the sum-of-products expression from the truth table of Figure 2.9: $Y = \overline{A}\,\overline{B} + A\overline{B}$. By Theorem T10, the equation simplifies to $Y = \overline{B}$. This may have been obvious looking at the truth table. In general, multiple steps may be necessary to simplify more complex equations.

The basic principle of simplifying sum-of-products equations is to combine terms using the relationship $PA + P\overline{A} = P$, where P may be any implicant. How far can an equation be simplified? We define an equation in sum-of-products form to be *minimized* if it uses the fewest possible implicants. If there are several equations with the same number of implicants, the minimal one is the one with the fewest literals.

An implicant is called a *prime implicant* if it cannot be combined with any other implicants in the equation to form a new implicant with fewer literals. The implicants in a minimal equation must all be prime implicants. Otherwise, they could be combined to reduce the number of literals.

Example 2.6 EQUATION MINIMIZATION

Minimize Equation 2.3: $\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$.

Solution: We start with the original equation and apply Boolean theorems step by step, as shown in Table 2.4.

Have we simplified the equation completely at this point? Let's take a closer look. From the original equation, the minterms $\overline{A}\,\overline{B}\,\overline{C}$ and $A\overline{B}\,\overline{C}$ differ only in the variable A. So we combined the minterms to form $\overline{B}\,\overline{C}$. However, if we look at the original equation, we note that the last two minterms $A\overline{B}\,\overline{C}$ and $A\overline{B}C$ also differ by a single literal (C and $\overline{C}$). Thus, using the same method, we could have combined these two minterms to form the minterm $A\overline{B}$. We say that implicants $\overline{B}\,\overline{C}$ and $A\overline{B}$ *share* the minterm $A\overline{B}\,\overline{C}$.

So, are we stuck with simplifying only one of the minterm pairs, or can we simplify both? Using the idempotency theorem, we can duplicate terms as many times as we want: $B = B + B + B + B \ldots$. Using this principle, we simplify the equation completely to its two prime implicants, $\overline{B}\,\overline{C} + A\overline{B}$, as shown in Table 2.5.

Table 2.4 **Equation minimization**

Step	Equation	Justification
	$\overline{A}\,\overline{B}\,\overline{C}+A\overline{B}\,\overline{C}+A\overline{B}C$	
1	$\overline{B}\,\overline{C}(\overline{A}+A)+A\overline{B}C$	T8: Distributivity
2	$\overline{B}\,\overline{C}(1)+A\overline{B}C$	T5: Complements
3	$\overline{B}\,\overline{C}+A\overline{B}C$	T1: Identity

Table 2.5 **Improved equation minimization**

Step	Equation	Justification
	$\overline{A}\,\overline{B}\,\overline{C}+A\overline{B}\,\overline{C}+A\overline{B}C$	
1	$\overline{A}\,\overline{B}\,\overline{C}+A\overline{B}\,\overline{C}+A\overline{B}\,\overline{C}+A\overline{B}C$	T3: Idempotency
2	$\overline{B}\,\overline{C}(\overline{A}+A)+A\overline{B}(\overline{C}+C)$	T8: Distributivity
3	$\overline{B}\,\overline{C}(1)+A\overline{B}(1)$	T5: Complements
4	$\overline{B}\,\overline{C}+A\overline{B}$	T1: Identity

Although it is a bit counterintuitive, *expanding* an implicant (for example, turning AB into $ABC+AB\overline{C}$) is sometimes useful in minimizing equations. By doing this, you can repeat one of the expanded minterms to be combined (shared) with another minterm.

You may have noticed that completely simplifying a Boolean equation with the theorems of Boolean algebra can take some trial and error. Section 2.7 describes a methodical technique called Karnaugh maps that makes the process easier.

Why bother simplifying a Boolean equation if it remains logically equivalent? Simplifying reduces the number of gates used to physically implement the function, thus making it smaller, cheaper, and possibly faster. The next section describes how to implement Boolean equations with logic gates.

2.4 FROM LOGIC TO GATES

A *schematic* is a diagram of a digital circuit showing the elements and the wires that connect them together. For example, the schematic in Figure 2.23 shows a possible hardware implementation of our favorite logic function, Equation 2.3:

$$Y=\overline{A}\,\overline{B}\,\overline{C}+A\overline{B}\,\overline{C}+A\overline{B}\,C.$$

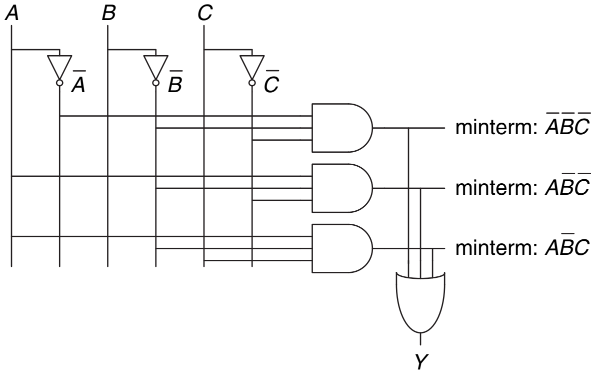


Figure 2.23 **Schematic of $Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$**

By drawing schematics in a consistent fashion, we make them easier to read and debug. We will generally obey the following guidelines:

- Inputs are on the left (or top) side of a schematic.
- Outputs are on the right (or bottom) side of a schematic.
- Whenever possible, gates should flow from left to right.
- Straight wires are better to use than wires with multiple corners (jagged wires waste mental effort following the wire rather than thinking of what the circuit does).
- Wires always connect at a T junction.
- A dot where wires cross indicates a connection between the wires.
- Wires crossing *without* a dot make no connection.

The last three guidelines are illustrated in Figure 2.24.

Any Boolean equation in sum-of-products form can be drawn as a schematic in a systematic way similar to Figure 2.23. First, draw columns for the inputs. Place inverters in adjacent columns to provide the complementary inputs if necessary. Draw rows of AND gates for each of the minterms. Then, for each output, draw an OR gate connected to the minterms related to that output. This style is called a *programmable logic array* (*PLA*) because the inverters, AND gates, and OR gates are arrayed in a systematic fashion. PLAs will be discussed further in Section 5.6.

Figure 2.25 shows an implementation of the simplified equation we found using Boolean algebra in Example 2.6. Notice that the simplified circuit has significantly less hardware than that of Figure 2.23. It may also be faster, because it uses gates with fewer inputs.

We can reduce the number of gates even further (albeit by a single inverter) by taking advantage of inverting gates. Observe that $\bar{B}\,\bar{C}$ is an

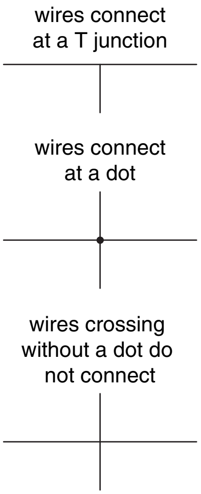


Figure 2.24 **Wire connections**

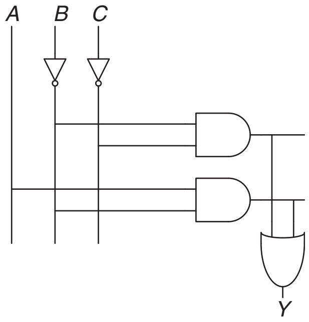


Figure 2.25 **Schematic of $Y = \bar{B}\bar{C} + A\bar{B}$**

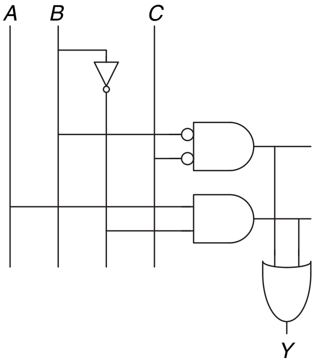


Figure 2.26 **Schematic using fewer gates**

AND with inverted inputs. Figure 2.26 shows a schematic using this optimization to eliminate the inverter on C. Recall that by De Morgan's theorem the AND with inverted inputs is equivalent to a NOR gate. Depending on the implementation technology, it may be cheaper to use the fewest gates or to use certain types of gates in preference to others. For example, NANDs and NORs are preferred over ANDs and ORs in CMOS implementations.

Many circuits have multiple outputs, each of which computes a separate Boolean function of the inputs. We can write a separate truth table for each output, but it is often convenient to write all of the outputs on a single truth table and sketch one schematic with all of the outputs.

Example 2.7 MULTIPLE-OUTPUT CIRCUITS

The dean, the department chair, the teaching assistant, and the dorm social chair each use the auditorium from time to time. Unfortunately, they occasionally conflict, leading to disasters such as the one that occurred when the dean's fundraising meeting with crusty trustees happened at the same time as the dorm's BTB[1] party. Alyssa P. Hacker has been called in to design a room reservation system.

The system has four inputs, $A_3, \ldots, A_0$, and four outputs, $Y_3, \ldots, Y_0$. These signals can also be written as $A_{3:0}$ and $Y_{3:0}$. Each user asserts her input when she requests the auditorium for the next day. The system asserts at most one output, granting the auditorium to the highest priority user. The dean, who is paying for the system, demands highest priority (3). The department chair, teaching assistant, and dorm social chair have decreasing priority.

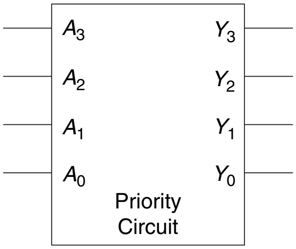
Write a truth table and Boolean equations for the system. Sketch a circuit that performs this function.

Solution: This function is called a four-input *priority circuit*. Its symbol and truth table are shown in Figure 2.27.

We could write each output in sum-of-products form and reduce the equations using Boolean algebra. However, the simplified equations are clear by inspection from the functional description (and the truth table): Y_3 is TRUE whenever A_3 is asserted, so $Y_3 = A_3$. Y_2 is TRUE if A_2 is asserted and A_3 is not asserted, so $Y_2 = \overline{A}_3 A_2$. Y_1 is TRUE if A_1 is asserted and neither of the higher priority inputs is asserted: $Y_1 = \overline{A}_3 \overline{A}_2 A_1$. And Y_0 is TRUE whenever A_0 and no other input is asserted: $Y_0 = \overline{A}_3 \overline{A}_2 \overline{A}_1 A_0$. The schematic is shown in Figure 2.28. An experienced designer can often implement a logic circuit by inspection. Given a clear specification, simply turn the words into equations and the equations into gates.

---

[1] Black light, twinkies, and beer.



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Figure 2.27 **Priority circuit**

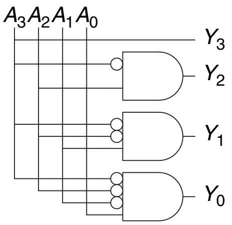


Figure 2.28 **Priority circuit schematic**

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Figure 2.29 **Priority circuit truth table with don't cares (X's)**

Notice that if A_3 is asserted in the priority circuit, the outputs *don't care* what the other inputs are. We use the symbol X to describe inputs that the output doesn't care about. Figure 2.29 shows that the four-input priority circuit truth table becomes much smaller with don't cares. From this truth table, we can easily read the Boolean equations in sum-of-products form by ignoring inputs with X's. Don't cares can also appear in truth table outputs, as we will see in Section 2.7.3.

X is an overloaded symbol that means "don't care" in truth tables and "contention" in logic simulation (see Section 2.6.1). Think about the context so you don't mix up the meanings. Some authors use D or ? instead for "don't care" to avoid this ambiguity.

2.5 MULTILEVEL COMBINATIONAL LOGIC

Logic in sum-of-products form is called *two-level logic* because it consists of literals connected to a level of AND gates connected to a level of OR gates. Designers often build circuits with more than two levels of logic gates. These multilevel combinational circuits may use less hardware than their two-level counterparts. Bubble pushing is especially helpful in analyzing and designing multilevel circuits.

2.5.1 Hardware Reduction

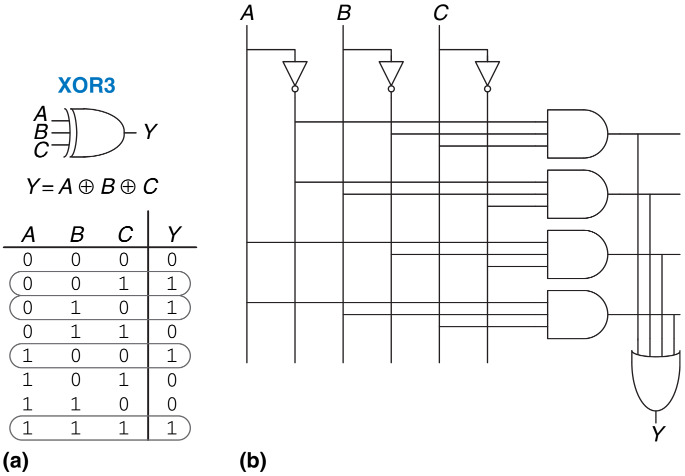
Some logic functions require an enormous amount of hardware when built using two-level logic. A notable example is the XOR function of multiple variables. For example, consider building a three-input XOR using the two-level techniques we have studied so far.

Recall that an N-input XOR produces a TRUE output when an odd number of inputs are TRUE. Figure 2.30 shows the truth table for a three-input XOR with the rows circled that produce TRUE outputs. From the truth table, we read off a Boolean equation in sum-of-products form in Equation 2.6. Unfortunately, there is no way to simplify this equation into fewer implicants.

$$Y = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC \tag{2.6}$$

On the other hand, $A \oplus B \oplus C = (A \oplus B) \oplus C$ (prove this to yourself by perfect induction if you are in doubt). Therefore, the three-input XOR can be built out of a cascade of two-input XORs, as shown in Figure 2.31.

Similarly, an eight-input XOR would require 128 eight-input AND gates and one 128-input OR gate for a two-level sum-of-products implementation. A much better option is to use a tree of two-input XOR gates, as shown in Figure 2.32.



A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 2.30 **Three-input XOR: (a) functional specification and (b) two-level logic implementation**

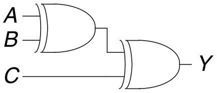


Figure 2.31 Three-input XOR using two-input XORs

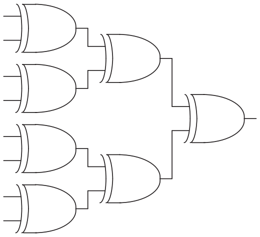


Figure 2.32 Eight-input XOR using seven two-input XORs

Selecting the best multilevel implementation of a specific logic function is not a simple process. Moreover, "best" has many meanings: fewest gates, fastest, shortest design time, least cost, least power consumption. In Chapter 5, you will see that the "best" circuit in one technology is not necessarily the best in another. For example, we have been using ANDs and ORs, but in CMOS, NANDs and NORs are more efficient. With some experience, you will find that you can create a good multilevel design by inspection for most circuits. You will develop some of this experience as you study circuit examples through the rest of this book. As you are learning, explore various design options and think about the trade-offs. Computer-aided design (CAD) tools are also available to search a vast space of possible multilevel designs and seek the one that best fits your constraints given the available building blocks.

2.5.2 Bubble Pushing

You may recall from Section 1.7.6 that CMOS circuits prefer NANDs and NORs over ANDs and ORs. But reading the equation by inspection from a multilevel circuit with NANDs and NORs can get pretty hairy. Figure 2.33 shows a multilevel circuit whose function is not immediately clear by inspection. Bubble pushing is a helpful way to redraw these circuits so that the bubbles cancel out and the function can be more easily determined. Building on the principles from Section 2.3.3, the guidelines for bubble pushing are as follows:

- Begin at the output of the circuit and work toward the inputs.
- Push any bubbles on the final output back toward the inputs so that you can read an equation in terms of the output (for example, Y) instead of the complement of the output ($\overline{Y}$).
- Working backward, draw each gate in a form so that bubbles cancel. If the current gate has an input bubble, draw the preceding gate with an output bubble. If the current gate does not have an input bubble, draw the preceding gate without an output bubble.

Figure 2.34 shows how to redraw Figure 2.33 according to the bubble pushing guidelines. Starting at the output Y, the NAND gate has a bubble on the output that we wish to eliminate. We push the output bubble back to form an OR with inverted inputs, shown in

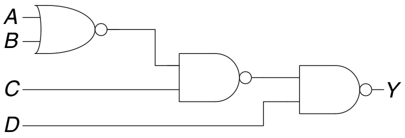


Figure 2.33 Multilevel circuit using NANDs and NORs

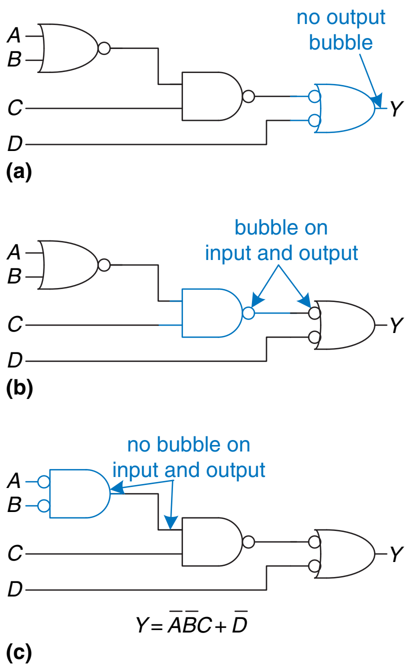

no output bubble
A
B
C
D
Y
(a)
bubble on input and output
A
B
C
D
Y
(b)
no bubble on input and output
A
B
C
D
Y
$Y = \overline{A}\,\overline{B}C + \overline{D}$
(c)

Figure 2.34 **Bubble-pushed circuit**

Figure 2.34(a). Working to the left, the rightmost gate has an input bubble that cancels with the output bubble of the middle NAND gate, so no change is necessary, as shown in Figure 2.34(b). The middle gate has no input bubble, so we transform the leftmost gate to have no output bubble, as shown in Figure 2.34(c). Now all of the bubbles in the circuit cancel except at the inputs, so the function can be read by inspection in terms of ANDs and ORs of true or complementary inputs: $Y = \overline{A}\,\overline{B}C + \overline{D}$.

For emphasis of this last point, Figure 2.35 shows a circuit logically equivalent to the one in Figure 2.34. The functions of internal nodes are labeled in blue. Because bubbles in series cancel, we can ignore the bubbles on the output of the middle gate and on one input of the rightmost gate to produce the logically equivalent circuit of Figure 2.35.

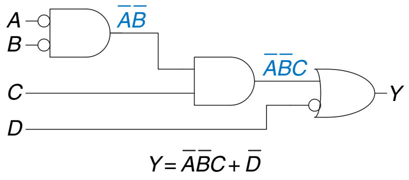

A
B
$\overline{A}\,\overline{B}$
C
D
$\overline{A}\,\overline{B}C$
Y
$Y = \overline{A}\,\overline{B}C + \overline{D}$

Figure 2.35 **Logically equivalent bubble-pushed circuit**

Example 2.8 BUBBLE PUSHING FOR CMOS LOGIC

Most designers think in terms of AND and OR gates, but suppose you would like to implement the circuit in Figure 2.36 in CMOS logic, which favors NAND and NOR gates. Use bubble pushing to convert the circuit to NANDs, NORs, and inverters.

Solution: A brute force solution is to just replace each AND gate with a NAND and an inverter, and each OR gate with a NOR and an inverter, as shown in Figure 2.37. This requires eight gates. Notice that the inverter is drawn with the bubble on the front rather than back, to emphasize how the bubble can cancel with the preceding inverting gate.

For a better solution, observe that bubbles can be added to the output of a gate and the input of the next gate without changing the function, as shown in Figure 2.38(a). The final AND is converted to a NAND and an inverter, as shown in Figure 2.38(b). This solution requires only five gates.

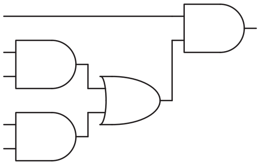


Figure 2.36 **Circuit using ANDs and ORs**

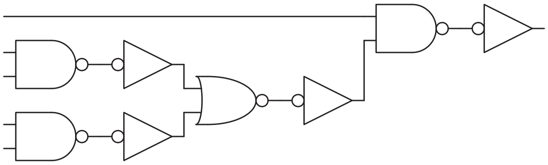


Figure 2.37 **Poor circuit using NANDs and NORs**

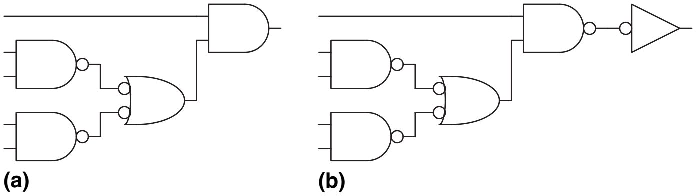


Figure 2.38 **Better circuit using NANDs and NORs**

2.6 X'S AND Z'S, OH MY

Boolean algebra is limited to 0's and 1's. However, real circuits can also have illegal and floating values, represented symbolically by X and Z.

2.6.1 Illegal Value: X

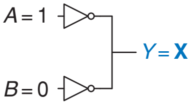


Figure 2.39 **Circuit with contention**

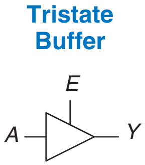
The symbol X indicates that the circuit node has an *unknown* or *illegal* value. This commonly happens if it is being driven to both 0 and 1 at the same time. Figure 2.39 shows a case where node Y is driven both HIGH and LOW. This situation, called *contention*, is considered to be an error and must be avoided. The actual voltage on a node with contention may be somewhere between 0 and V_{DD}, depending on the relative strengths of the gates driving HIGH and LOW. It is often, but not always, in the forbidden zone. Contention also can cause large amounts of power to flow between the fighting gates, resulting in the circuit getting hot and possibly damaged.

X values are also sometimes used by circuit simulators to indicate an uninitialized value. For example, if you forget to specify the value of an input, the simulator may assume it is an X to warn you of the problem.

As mentioned in Section 2.4, digital designers also use the symbol X to indicate "don't care" values in truth tables. Be sure not to mix up the two meanings. When X appears in a truth table, it indicates that the value of the variable in the truth table is unimportant (can be either 0 or 1). When X appears in a circuit, it means that the circuit node has an unknown or illegal value.

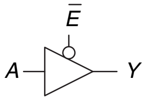
2.6.2 Floating Value: Z

The symbol Z indicates that a node is being driven neither HIGH nor LOW. The node is said to be *floating*, *high impedance*, or *high* Z. A typical misconception is that a floating or undriven node is the same as a logic 0. In reality, a floating node might be 0, might be 1, or might be at some voltage in between, depending on the history of the system. A floating node does not always mean there is an error in the circuit, so long as some other circuit element does drive the node to a valid logic level when the value of the node is relevant to circuit operation.



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Figure 2.40 Tristate buffer



$\overline{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

Figure 2.41 Tristate buffer with active low enable

One common way to produce a floating node is to forget to connect a voltage to a circuit input, or to assume that an unconnected input is the same as an input with the value of 0. This mistake may cause the circuit to behave erratically as the floating input randomly changes from 0 to 1. Indeed, touching the circuit may be enough to trigger the change by means of static electricity from the body. We have seen circuits that operate correctly only as long as the student keeps a finger pressed on a chip.

The *tristate buffer*, shown in Figure 2.40, has three possible output states: HIGH (1), LOW (0), and floating (Z). The tristate buffer has an input A, output Y, and *enable* E. When the enable is TRUE, the tristate buffer acts as a simple buffer, transferring the input value to the output. When the enable is FALSE, the output is allowed to float (Z).

The tristate buffer in Figure 2.40 has an *active high* enable. That is, when the enable is HIGH (1), the buffer is enabled. Figure 2.41 shows a tristate buffer with an *active low* enable. When the enable is LOW (0),

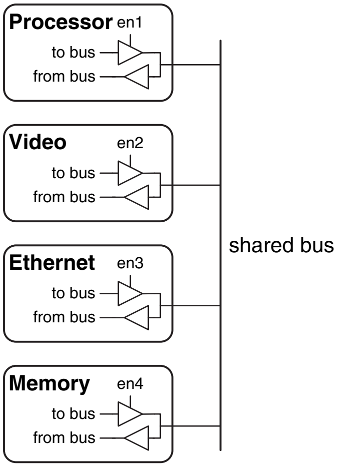


Figure 2.42 **Tristate bus connecting multiple chips**

the buffer is enabled. We show that the signal is active low by putting a bubble on its input wire. We often indicate an active low input by drawing a bar over its name, $\overline{E}$, or appending the letters "b" or "bar" after its name, *Eb* or *Ebar*.

Tristate buffers are commonly used on *busses* that connect multiple chips. For example, a microprocessor, a video controller, and an Ethernet controller might all need to communicate with the memory system in a personal computer. Each chip can connect to a shared memory bus using tristate buffers, as shown in Figure 2.42. Only one chip at a time is allowed to assert its enable signal to drive a value onto the bus. The other chips must produce floating outputs so that they do not cause contention with the chip talking to the memory. Any chip can read the information from the shared bus at any time. Such tristate busses were once common. However, in modern computers, higher speeds are possible with *point-to-point links*, in which chips are connected to each other directly rather than over a shared bus.

2.7 KARNAUGH MAPS

After working through several minimizations of Boolean equations using Boolean algebra, you will realize that, if you're not careful, you sometimes end up with a completely *different* equation instead of a simplified equation. *Karnaugh maps* (*K-maps*) are a graphical method for simplifying Boolean equations. They were invented in 1953 by Maurice Karnaugh, a telecommunications engineer at Bell Labs. K-maps work well for problems

Maurice Karnaugh, 1924–. Graduated with a bachelor's degree in physics from the City College of New York in 1948 and earned a Ph.D. in physics from Yale in 1952.

Worked at Bell Labs and IBM from 1952 to 1993 and as a computer science professor at the Polytechnic University of New York from 1980 to 1999.

Gray codes were patented (U.S. Patent 2,632,058) by Frank Gray, a Bell Labs researcher, in 1953. They are especially useful in mechanical encoders because a slight misalignment causes an error in only one bit.

Gray codes generalize to any number of bits. For example, a 3-bit Gray code sequence is:

```
000, 001, 011, 010,
110, 111, 101, 100
```

Lewis Carroll posed a related puzzle in *Vanity Fair* in 1879.

> "The rules of the Puzzle are simple enough. Two words are proposed, of the same length; and the puzzle consists of linking these together by interposing other words, each of which shall differ from the next word in one letter only. That is to say, one letter may be changed in one of the given words, then one letter in the word so obtained, and so on, till we arrive at the other given word."

For example, SHIP to DOCK:

```
SHIP, SLIP, SLOP,
SLOT, SOOT, LOOT,
LOOK, LOCK, DOCK.
```

Can you find a shorter sequence?

with up to four variables. More important, they give insight into manipulating Boolean equations.

Recall that logic minimization involves combining terms. Two terms containing an implicant P and the true and complementary forms of some variable A are combined to eliminate A: $PA + P\overline{A} = P$. Karnaugh maps make these combinable terms easy to see by putting them next to each other in a grid.

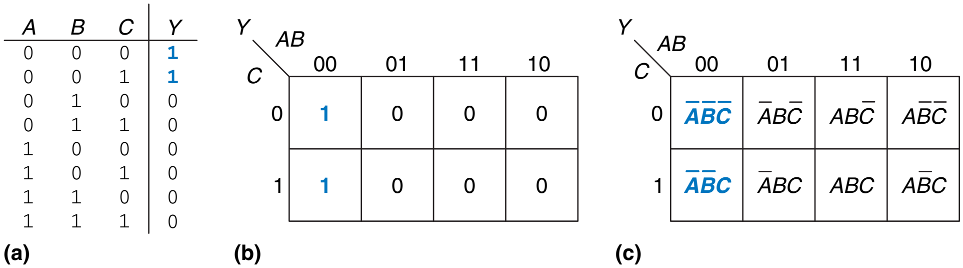
Figure 2.43 shows the truth table and K-map for a three-input function. The top row of the K-map gives the four possible values for the A and B inputs. The left column gives the two possible values for the C input. Each square in the K-map corresponds to a row in the truth table and contains the value of the output Y for that row. For example, the top left square corresponds to the first row in the truth table and indicates that the output value $Y = 1$ when $ABC = 000$. Just like each row in a truth table, each square in a K-map represents a single minterm. For the purpose of explanation, Figure 2.43(c) shows the minterm corresponding to each square in the K-map.

Each square, or minterm, differs from an adjacent square by a change in a single variable. This means that adjacent squares share all the same literals except one, which appears in true form in one square and in complementary form in the other. For example, the squares representing the minterms $\overline{A}\,\overline{B}\,\overline{C}$ and $\overline{A}\,\overline{B}C$ are adjacent and differ only in the variable C. You may have noticed that the A and B combinations in the top row are in a peculiar order: 00, 01, 11, 10. This order is called a *Gray code*. It differs from ordinary binary order (00, 01, 10, 11) in that adjacent entries differ only in a single variable. For example, 01 : 11 only changes A from 0 to 1, while 01 : 10 would change A from 1 to 0 and B from 0 to 1. Hence, writing the combinations in binary order would not have produced our desired property of adjacent squares differing only in one variable.

The K-map also "wraps around." The squares on the far right are effectively adjacent to the squares on the far left, in that they differ only in one variable, A. In other words, you could take the map and roll it into a cylinder, then join the ends of the cylinder to form a torus (i.e., a donut), and still guarantee that adjacent squares would differ only in one variable.

2.7.1 Circular Thinking

In the K-map in Figure 2.43, only two minterms are present in the equation, $\overline{A}\,\overline{B}\,\overline{C}$ and $\overline{A}\,\overline{B}C$, as indicated by the 1's in the left column. Reading the minterms from the K-map is exactly equivalent to reading equations in sum-of-products form directly from the truth table.



A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

(a)

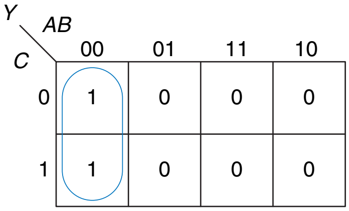
Y: C \ AB	00	01	11	10
0	1	0	0	0
1	1	0	0	0

(b)

Y: C \ AB	00	01	11	10
0	$\overline{A}\,\overline{B}\,\overline{C}$	$\overline{A}B\overline{C}$	$AB\overline{C}$	$A\overline{B}\,\overline{C}$
1	$\overline{A}\,\overline{B}C$	$\overline{A}BC$	ABC	$A\overline{B}C$

(c)

Figure 2.43 Three-input function: (a) truth table, (b) K-map, (c) K-map showing minterms



Y: C \ AB	00	01	11	10
0	1	0	0	0
1	1	0	0	0

Figure 2.44 K-map minimization

As before, we can use Boolean algebra to minimize equations in sum-of-products form.

$$Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C = \overline{A}\,\overline{B}(\overline{C} + C) = \overline{A}\,\overline{B} \tag{2.7}$$

K-maps help us do this simplification graphically by *circling* 1's in adjacent squares, as shown in Figure 2.44. For each circle, we write the corresponding implicant. Remember from Section 2.2 that an implicant is the product of one or more literals. Variables whose true *and* complementary forms are both in the circle are excluded from the implicant. In this case, the variable C has both its true form (1) and its complementary form (0) in the circle, so we do not include it in the implicant. In other words, Y is TRUE when $A = B = 0$, independent of C. So the implicant is $\overline{A}\,\overline{B}$. The K-map gives the same answer we reached using Boolean algebra.

2.7.2 Logic Minimization with K-Maps

K-maps provide an easy visual way to minimize logic. Simply circle all the rectangular blocks of 1's in the map, using the fewest possible number of circles. Each circle should be as large as possible. Then read off the implicants that were circled.

More formally, recall that a Boolean equation is minimized when it is written as a sum of the fewest number of prime implicants. Each circle on the K-map represents an implicant. The largest possible circles are prime implicants.

For example, in the K-map of Figure 2.44, $\overline{A}\,\overline{B}\,\overline{C}$ and $\overline{A}\,\overline{B}C$ are implicants, but *not* prime implicants. Only $\overline{A}\,\overline{B}$ is a prime implicant in that K-map. Rules for finding a minimized equation from a K-map are as follows:

- Use the fewest circles necessary to cover all the 1's.
- All the squares in each circle must contain 1's.
- Each circle must span a rectangular block that is a power of 2 (i.e., 1, 2, or 4) squares in each direction.
- Each circle should be as large as possible.
- A circle may wrap around the edges of the K-map.
- A 1 in a K-map may be circled multiple times if doing so allows fewer circles to be used.

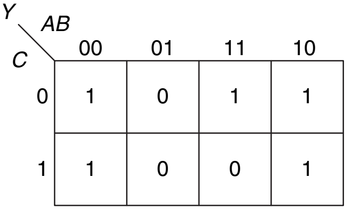
Example 2.9 MINIMIZATION OF A THREE-VARIABLE FUNCTION USING A K-MAP

Suppose we have the function $Y = F(A, B, C)$ with the K-map shown in Figure 2.45. Minimize the equation using the K-map.

Solution: Circle the 1's in the K-map using as few circles as possible, as shown in Figure 2.46. Each circle in the K-map represents a prime implicant, and the dimension of each circle is a power of two (2×1 and 2×2). We form the prime implicant for each circle by writing those variables that appear in the circle only in true or only in complementary form.

For example, in the 2×1 circle, the true and complementary forms of B are included in the circle, so we *do not* include B in the prime implicant. However, only the true form of A (A) and complementary form of C ($\overline{C}$) are in this circle, so we include these variables in the prime implicant $A\overline{C}$. Similarly, the 2×2 circle covers all squares where $B = 0$, so the prime implicant is $\overline{B}$.

Notice how the top-right square (minterm) is covered twice to make the prime implicant circles as large as possible. As we saw with Boolean algebra techniques, this is equivalent to sharing a minterm to reduce the size of the implicant. Also notice how the circle covering four squares wraps around the sides of the K-map.



Y C \ AB	00	01	11	10
0	1	0	1	1
1	1	0	0	1

Figure 2.45 K-map for Example 2.9

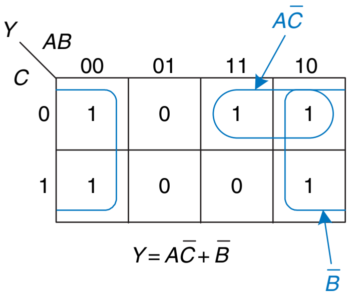


Figure 2.46 **Solution for Example 2.9**

Example 2.10 SEVEN-SEGMENT DISPLAY DECODER

A *seven-segment display decoder* takes a 4-bit data input $D_{3:0}$ and produces seven outputs to control light-emitting diodes to display a digit from 0 to 9. The seven outputs are often called segments *a* through *g*, or S_a–S_g, as defined in Figure 2.47. The digits are shown in Figure 2.48. Write a truth table for the outputs, and use K-maps to find Boolean equations for outputs S_a and S_b. Assume that illegal input values (10–15) produce a blank readout.

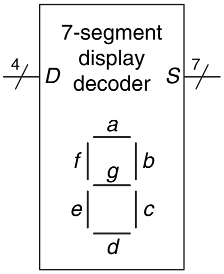


Figure 2.47 **Seven-segment display decoder icon**

Solution: The truth table is given in Table 2.6. For example, an input of 0000 should turn on all segments except S_g.

Each of the seven outputs is an independent function of four variables. The K-maps for outputs S_a and S_b are shown in Figure 2.49. Remember that adjacent squares may differ in only a single variable, so we label the rows and columns in Gray code order: 00, 01, 11, 10. Be careful to also remember this ordering when *entering* the output values into the squares.

Next, circle the prime implicants. Use the fewest number of circles necessary to cover all the 1's. A circle can wrap around the edges (vertical *and* horizontal), and a 1 may be circled more than once. Figure 2.50 shows the prime implicants and the simplified Boolean equations.

Note that the minimal set of prime implicants is not unique. For example, the 0000 entry in the S_a K-map was circled along with the 1000 entry to produce the $\overline{D}_2\overline{D}_1\overline{D}_0$ minterm. The circle could have included the 0010 entry instead, producing a $\overline{D}_3\overline{D}_2\overline{D}_0$ minterm, as shown with dashed lines in Figure 2.51.

Figure 2.52 (see page 82) illustrates a common error in which a nonprime implicant was chosen to cover the 1 in the upper left corner. This minterm, $\overline{D}_3\overline{D}_2\overline{D}_1\overline{D}_0$, gives a sum-of-products equation that is *not* minimal. The minterm could have been combined with either of the adjacent ones to form a larger circle, as was done in the previous two figures.

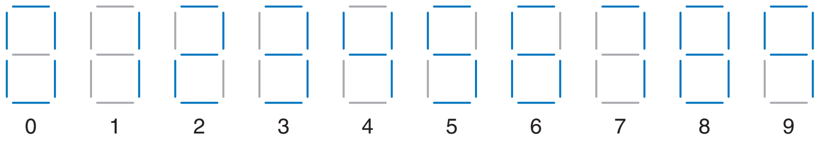
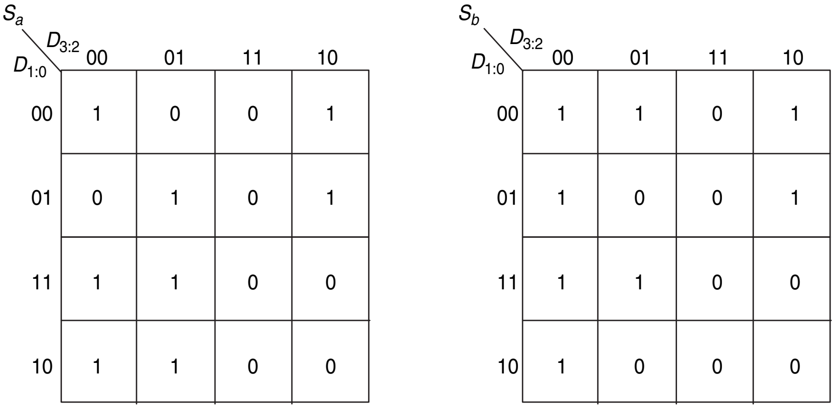


Figure 2.48 Seven-segment display digits

Table 2.6 Seven-segment display decoder truth table

$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	0	0	0	0	0	0



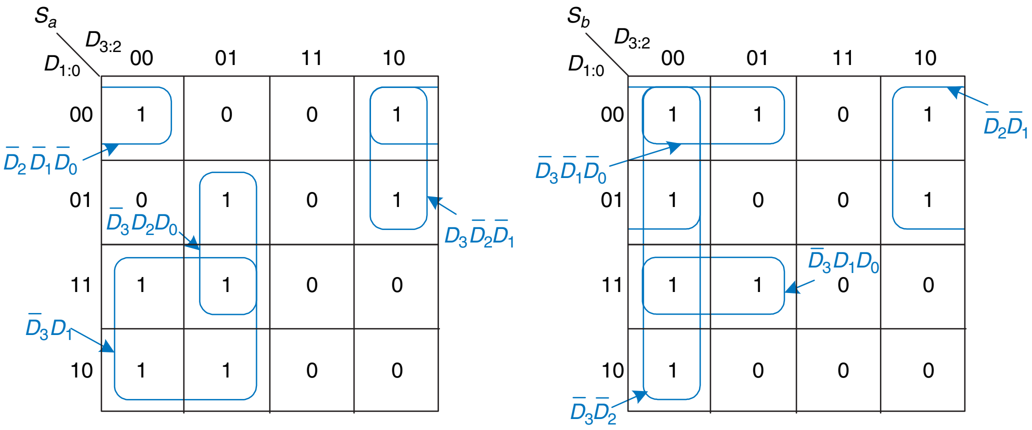
S_a

$D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	0	0	1
01	0	1	0	1
11	1	1	0	0
10	1	1	0	0

S_b

$D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	1	0	1
01	1	0	0	1
11	1	1	0	0
10	1	0	0	0

Figure 2.49 Karnaugh maps for S_a and S_b



S_a
$D_{3:2}$
$D_{1:0}$
$\overline{D_2}\,\overline{D_1}\,\overline{D_0}$
$\overline{D_3}D_2D_0$
$D_3\overline{D_2}\,\overline{D_1}$
$\overline{D_3}D_1$
S_b
$D_{3:2}$
$D_{1:0}$
$\overline{D_2}\,\overline{D_1}$
$\overline{D_3}\,\overline{D_1}\,\overline{D_0}$
$\overline{D_3}D_1D_0$
$\overline{D_3}\,\overline{D_2}$

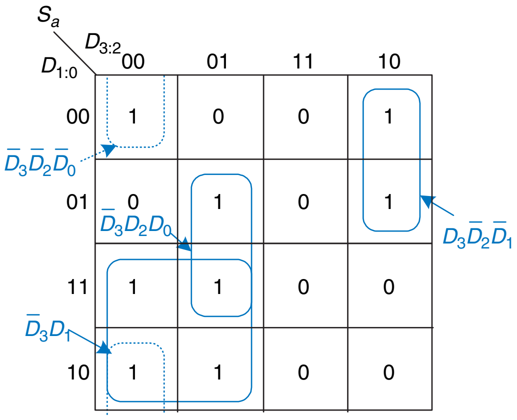
S_a: $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	0	0	1
01	0	1	0	1
11	1	1	0	0
10	1	1	0	0

S_b: $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	1	0	1
01	1	0	0	1
11	1	1	0	0
10	1	0	0	0

$$S_a = \overline{D_3}D_1 + \overline{D_3}D_2D_0 + D_3\overline{D_2}\,\overline{D_1} + \overline{D_2}\,\overline{D_1}\,\overline{D_0}$$

$$S_b = \overline{D_3}\,\overline{D_2} + \overline{D_2}\,\overline{D_1} + \overline{D_3}D_1D_0 + \overline{D_3}\,\overline{D_1}\,\overline{D_0}$$

Figure 2.50 K-map solution for Example 2.10



S_a
$D_{3:2}$
$D_{1:0}$
$\overline{D_3}\,\overline{D_2}\,\overline{D_0}$
$\overline{D_3}D_2D_0$
$D_3\overline{D_2}\,\overline{D_1}$
$\overline{D_3}D_1$

S_a: $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	0	0	1
01	0	1	0	1
11	1	1	0	0
10	1	1	0	0

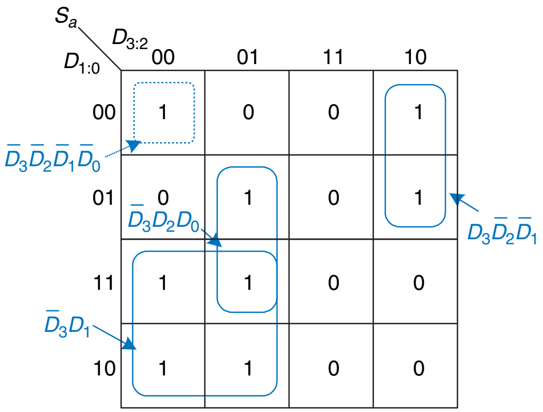
$$S_a = \overline{D_3}D_1 + \overline{D_3}D_2D_0 + D_3\overline{D_2}\,\overline{D_1} + \overline{D_3}\,\overline{D_2}\,\overline{D_0}$$

Figure 2.51 Alternative K-map for S_a showing different set of prime implicants

2.7.3 Don't Cares

Recall that "don't care" entries for truth table inputs were introduced in Section 2.4 to reduce the number of rows in the table when some variables do not affect the output. They are indicated by the symbol X, which means that the entry can be either 0 or 1.

Don't cares also appear in truth table outputs where the output value is unimportant or the corresponding input combination can never



$$S_a = \overline{D}_3D_1 + \overline{D}_3D_2D_0 + D_3\overline{D}_2\overline{D}_1 + \overline{D}_3\overline{D}_2\overline{D}_1\overline{D}_0$$

Figure 2.52 **Alternative K-map for S_a showing incorrect nonprime implicant**

happen. Such outputs can be treated as either 0's or 1's at the designer's discretion.

In a K-map, X's allow for even more logic minimization. They can be circled if they help cover the 1's with fewer or larger circles, but they do not have to be circled if they are not helpful.

Example 2.11 SEVEN-SEGMENT DISPLAY DECODER WITH DON'T CARES

Repeat Example 2.10 if we don't care about the output values for illegal input values of 10 to 15.

Solution: The K-map is shown in Figure 2.53 with X entries representing don't care. Because don't cares can be 0 or 1, we circle a don't care if it allows us to cover the 1's with fewer or bigger circles. Circled don't cares are treated as 1's, whereas uncircled don't cares are 0's. Observe how a 2 × 2 square wrapping around all four corners is circled for segment S_a. Use of don't cares simplifies the logic substantially.

2.7.4 The Big Picture

Boolean algebra and Karnaugh maps are two methods of logic simplification. Ultimately, the goal is to find a low-cost method of implementing a particular logic function.

In modern engineering practice, computer programs called *logic synthesizers* produce simplified circuits from a description of the logic function, as we will see in Chapter 4. For large problems, logic synthesizers are much more efficient than humans. For small problems, a

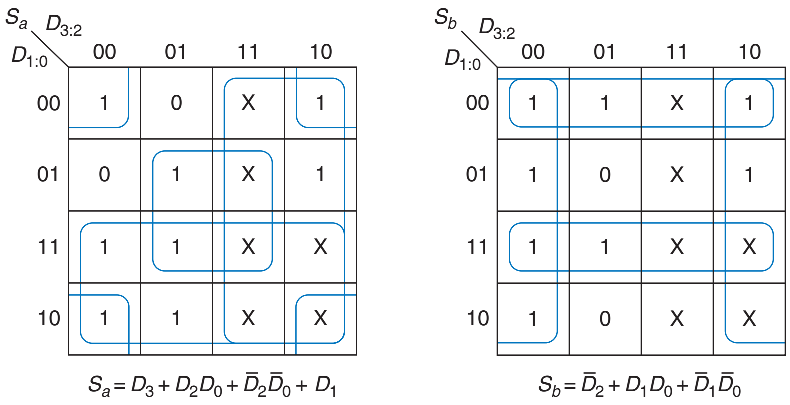

S_a
$D_{3:2}$
$D_{1:0}$
00
01
11
10
$S_a = D_3 + D_2D_0 + \overline{D}_2\overline{D}_0 + D_1$
S_b
$S_b = \overline{D}_2 + D_1D_0 + \overline{D}_1\overline{D}_0$

Figure 2.53 **K-map solution with don't cares**

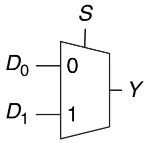
human with a bit of experience can find a good solution by inspection. Neither of the authors has ever used a Karnaugh map in real life to solve a practical problem. But the insight gained from the principles underlying Karnaugh maps is valuable. And Karnaugh maps often appear at job interviews!

2.8 COMBINATIONAL BUILDING BLOCKS

Combinational logic is often grouped into larger building blocks to build more complex systems. This is an application of the principle of abstraction, hiding the unnecessary gate-level details to emphasize the function of the building block. We have already studied three such building blocks: full adders (from Section 2.1), priority circuits (from Section 2.4), and seven-segment display decoders (from Section 2.7). This section introduces two more commonly used building blocks: multiplexers and decoders. Chapter 5 covers other combinational building blocks.

2.8.1 Multiplexers

Multiplexers are among the most commonly used combinational circuits. They choose an output from among several possible inputs based on the value of a *select* signal. A multiplexer is sometimes affectionately called a *mux*.



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 2.54 **2:1 multiplexer symbol and truth table**

2:1 Multiplexer

Figure 2.54 shows the schematic and truth table for a 2:1 multiplexer with two data inputs D_0 and D_1, a select input S, and one output Y. The multiplexer chooses between the two data inputs based on the select: if $S = 0$, $Y = D_0$, and if $S = 1$, $Y = D_1$. S is also called a *control signal* because it controls what the multiplexer does.

A 2:1 multiplexer can be built from sum-of-products logic as shown in Figure 2.55. The Boolean equation for the multiplexer may be derived

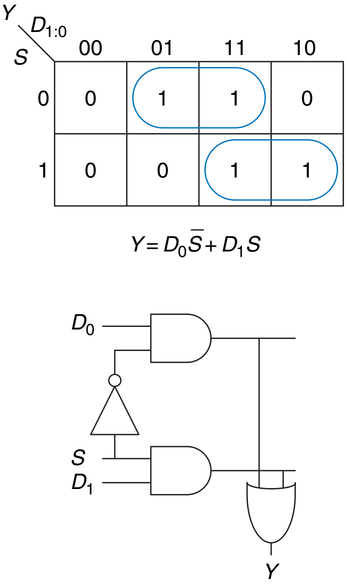


Figure 2.55 **2:1 multiplexer implementation using two-level logic**

with a Karnaugh map or read off by inspection (Y is 1 if $S = 0$ AND D_0 is 1 OR if $S = 1$ AND D_1 is 1).

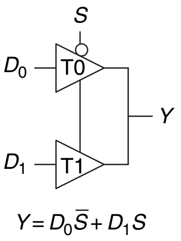


Figure 2.56 **Multiplexer using tristate buffers**

Alternatively, multiplexers can be built from tristate buffers as shown in Figure 2.56. The tristate enables are arranged such that, at all times, exactly one tristate buffer is active. When $S = 0$, tristate T0 is enabled, allowing D_0 to flow to Y. When $S = 1$, tristate T1 is enabled, allowing D_1 to flow to Y.

Wider Multiplexers

A 4:1 multiplexer has four data inputs and one output, as shown in Figure 2.57. Two select signals are needed to choose among the four data inputs. The 4:1 multiplexer can be built using sum-of-products logic, tristates, or multiple 2:1 multiplexers, as shown in Figure 2.58.

The product terms enabling the tristates can be formed using AND gates and inverters. They can also be formed using a decoder, which we will introduce in Section 2.8.2.

Wider multiplexers, such as 8:1 and 16:1 multiplexers, can be built by expanding the methods shown in Figure 2.58. In general, an N:1 multiplexer needs $\log_2 N$ select lines. Again, the best implementation choice depends on the target technology.

Shorting together the outputs of multiple gates technically violates the rules for combinational circuits given in Section 2.1. But because exactly one of the outputs is driven at any time, this exception is allowed.

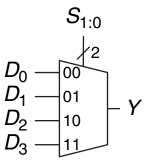


Figure 2.57 **4:1 multiplexer**

Multiplexer Logic

Multiplexers can be used as *lookup tables* to perform logic functions. Figure 2.59 shows a 4:1 multiplexer used to implement a two-input

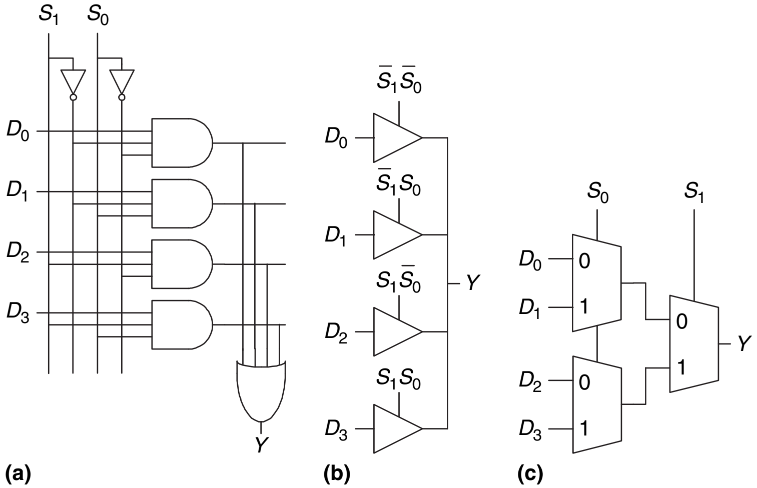


Figure 2.58 **4:1 multiplexer implementations: (a) two-level logic, (b) tristates, (c) hierarchical**

AND gate. The inputs, A and B, serve as select lines. The multiplexer data inputs are connected to 0 or 1 according to the corresponding row of the truth table. In general, a 2^N-input multiplexer can be programmed to perform any N-input logic function by applying 0's and 1's to the appropriate data inputs. Indeed, by changing the data inputs, the multiplexer can be reprogrammed to perform a different function.

With a little cleverness, we can cut the multiplexer size in half, using only a 2^{N-1}-input multiplexer to perform any N-input logic function. The strategy is to provide one of the literals, as well as 0's and 1's, to the multiplexer data inputs.

To illustrate this principle, Figure 2.60 shows two-input AND and XOR functions implemented with 2:1 multiplexers. We start with an ordinary truth table, and then combine pairs of rows to eliminate the rightmost input variable by expressing the output in terms of this variable. For example, in the case of AND, when $A = 0$, $Y = 0$, regardless of B. When $A = 1$, $Y = 0$ if $B = 0$ and $Y = 1$ if $B = 1$, so $Y = B$. We then use the multiplexer as a lookup table according to the new, smaller truth table.

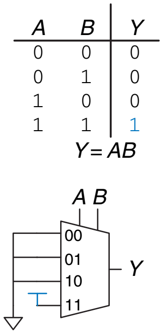


Figure 2.59 **4:1 multiplexer implementation of two-input AND function**

Example 2.12 LOGIC WITH MULTIPLEXERS

Alyssa P. Hacker needs to implement the function $Y = A\overline{B} + \overline{B}\,\overline{C} + \overline{A}BC$ to finish her senior project, but when she looks in her lab kit, the only part she has left is an 8:1 multiplexer. How does she implement the function?

Solution: Figure 2.61 shows Alyssa's implementation using a single 8:1 multiplexer. The multiplexer acts as a lookup table where each row in the truth table corresponds to a multiplexer input.

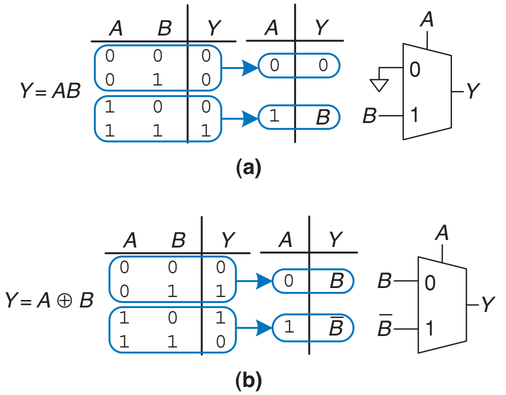

Y = AB
A B Y
0 0 0
0 1 0
1 0 0
1 1 1
A Y
0 0
1 B
A
0
1
B
Y
(a)
Y = A ⊕ B
A B Y
0 0 0
0 1 1
1 0 1
1 1 0
A Y
0 B
1 B̄
A
B 0
B̄ 1
Y
(b)

Figure 2.60 **Multiplexer logic using variable inputs**

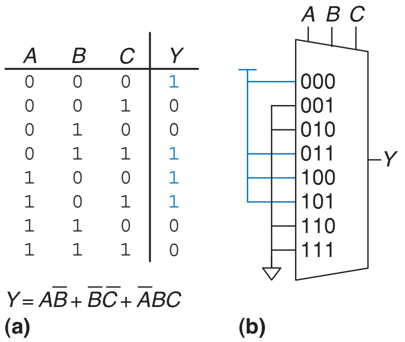

A B C Y
0 0 0 1
0 0 1 0
0 1 0 0
0 1 1 1
1 0 0 1
1 0 1 1
1 1 0 0
1 1 1 0
Y = AB̄ + B̄C̄ + ĀBC
(a)
A B C
000
001
010
011
100
101
110
111
Y
(b)

Figure 2.61 **Alyssa's circuit: (a) truth table, (b) 8:1 multiplexer implementation**

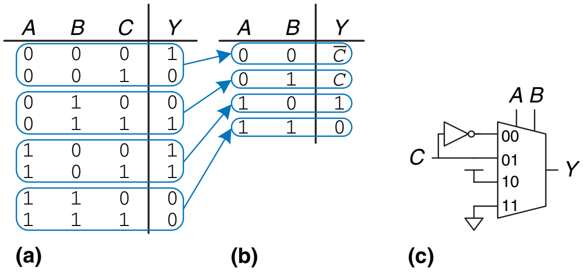
Example 2.13 LOGIC WITH MULTIPLEXERS, REPRISED

Alyssa turns on her circuit one more time before the final presentation and blows up the 8:1 multiplexer. (She accidently powered it with 20 V instead of 5 V after not sleeping all night.) She begs her friends for spare parts and they give her a 4:1 multiplexer and an inverter. Can she build her circuit with only these parts?

Solution: Alyssa reduces her truth table to four rows by letting the output depend on C. (She could also have chosen to rearrange the columns of the truth table to let the output depend on A or B.) Figure 2.62 shows the new design.

2.8.2 Decoders

A decoder has N inputs and 2^N outputs. It asserts exactly one of its outputs depending on the input combination. Figure 2.63 shows a 2:4 decoder. When $A_{1:0} = 00$, Y_0 is 1. When $A_{1:0} = 01$, Y_1 is 1. And so forth. The outputs are called *one-hot*, because exactly one is "hot" (HIGH) at a given time.



A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

(a)

A	B	Y
0	0	$\overline{C}$
0	1	C
1	0	1
1	1	0

(b)

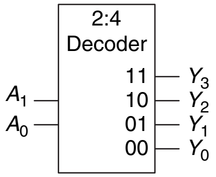
A B
C
00
01
10
11
Y
(c)

Figure 2.62 Alyssa's new circuit

Example 2.14 DECODER IMPLEMENTATION

Implement a 2:4 decoder with AND, OR, and NOT gates.

Solution: Figure 2.64 shows an implementation for the 2:4 decoder using four AND gates. Each gate depends on either the true or the complementary form of each input. In general, an $N{:}2^N$ decoder can be constructed from 2^N N-input AND gates that accept the various combinations of true or complementary inputs. Each output in a decoder represents a single minterm. For example, Y_0 represents the minterm $\overline{A_1}\,\overline{A_0}$. This fact will be handy when using decoders with other digital building blocks.


2:4
Decoder
A_1
A_0
11
10
01
00
Y_3
Y_2
Y_1
Y_0

A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Figure 2.63 2:4 decoder

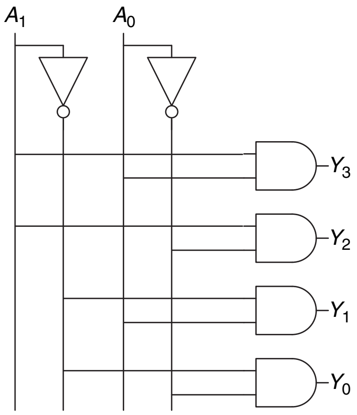

A_1
A_0
Y_3
Y_2
Y_1
Y_0

Figure 2.64 2:4 decoder implementation

Decoder Logic

Decoders can be combined with OR gates to build logic functions. Figure 2.65 shows the two-input XNOR function using a 2:4 decoder and a single OR gate. Because each output of a decoder represents a single minterm, the function is built as the OR of all the minterms in the function. In Figure 2.65, $Y = \overline{A}\,\overline{B} + AB = \overline{A \oplus B}$.

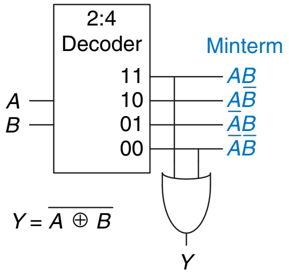

2:4
Decoder
A
B
11
10
01
00
Minterm
AB
$A\overline{B}$
$\overline{A}B$
$\overline{A}\,\overline{B}$
$Y = \overline{A \oplus B}$
Y

Figure 2.65 Logic function using decoder

When using decoders to build logic, it is easiest to express functions as a truth table or in canonical sum-of-products form. An N-input function with M 1's in the truth table can be built with an N:2^N decoder and an M-input OR gate attached to all of the minterms containing 1's in the truth table. This concept will be applied to the building of Read Only Memories (ROMs) in Section 5.5.6.

2.9 TIMING

In previous sections, we have been concerned primarily with whether the circuit works—ideally, using the fewest gates. However, as any seasoned circuit designer will attest, one of the most challenging issues in circuit design is *timing:* making a circuit run fast.

An output takes time to change in response to an input change. Figure 2.66 shows the *delay* between an input change and the subsequent output change for a buffer. The figure is called a *timing diagram;* it portrays the *transient response* of the buffer circuit when an input changes. The transition from LOW to HIGH is called the *rising edge*. Similarly, the transition from HIGH to LOW (not shown in the figure) is called the *falling edge*. The blue arrow indicates that the rising edge of Y is caused by the rising edge of A. We measure delay from the *50% point* of the input signal, A, to the 50% point of the output signal, Y. The 50% point is the point at which the signal is half-way (50%) between its LOW and HIGH values as it transitions.

When designers speak of calculating the *delay* of a circuit, they generally are referring to the worst-case value (the propagation delay), unless it is clear otherwise from the context.

2.9.1 Propagation and Contamination Delay

Combinational logic is characterized by its *propagation delay* and *contamination delay*. The propagation delay t_{pd} is the maximum time from when an input changes until the output or outputs reach their final value. The contamination delay t_{cd} is the minimum time from when an input changes until any output starts to change its value.

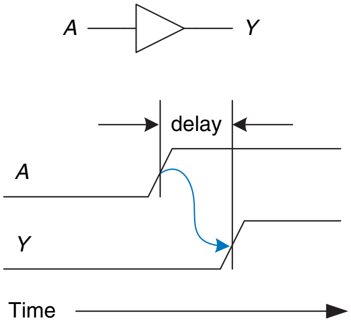


Figure 2.66 Circuit delay

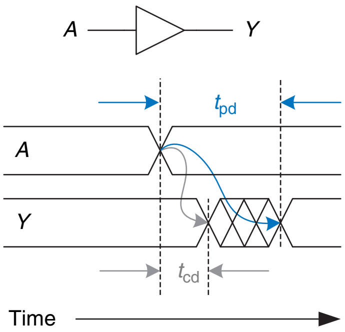


Figure 2.67 **Propagation and contamination delay**

Figure 2.67 illustrates a buffer's propagation delay and contamination delay in blue and gray, respectively. The figure shows that A is initially either HIGH or LOW and changes to the other state at a particular time; we are interested only in the fact that it changes, not what value it has. In response, Y changes some time later. The arcs indicate that Y may start to change t_{cd} after A transitions and that Y definitely settles to its new value within t_{pd}.

The underlying causes of delay in circuits include the time required to charge the capacitance in a circuit and the speed of light. t_{pd} and t_{cd} may be different for many reasons, including

- different rising and falling delays
- multiple inputs and outputs, some of which are faster than others
- circuits slowing down when hot and speeding up when cold

Circuit delays are ordinarily on the order of picoseconds (1 ps = 10^{-12} seconds) to nanoseconds (1 ns = 10^{-9} seconds). Trillions of picoseconds have elapsed in the time you spent reading this sidebar.

Calculating t_{pd} and t_{cd} requires delving into the lower levels of abstraction beyond the scope of this book. However, manufacturers normally supply data sheets specifying these delays for each gate.

Along with the factors already listed, propagation and contamination delays are also determined by the *path* a signal takes from input to output. Figure 2.68 shows a four-input logic circuit. The *critical path*, shown in blue, is the path from input A or B to output Y. It is the longest, and

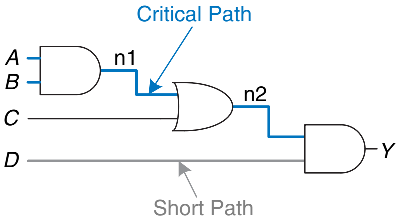


Figure 2.68 **Short path and critical path**

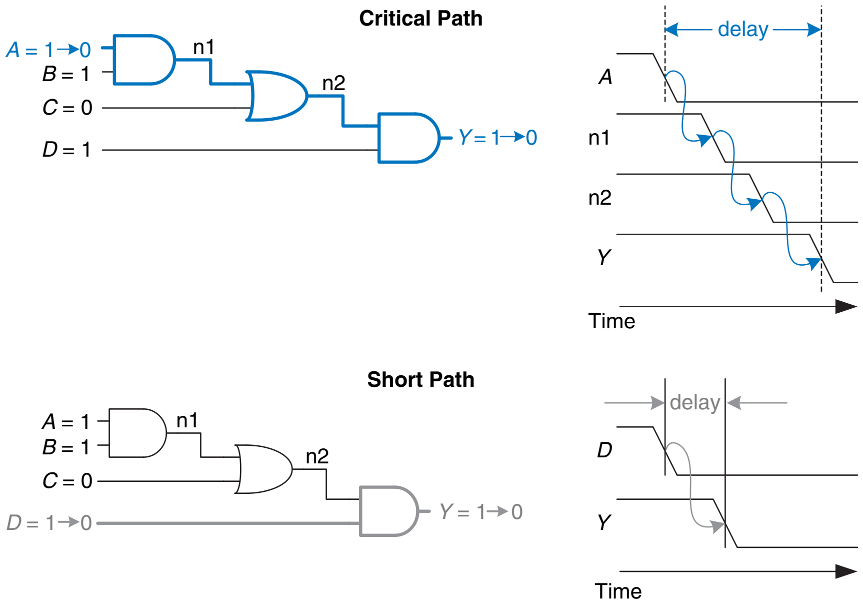


Figure 2.69 Critical and short path waveforms

therefore the slowest, path, because the input travels through three gates to the output. This path is critical because it limits the speed at which the circuit operates. The *short path* through the circuit, shown in gray, is from input *D* to output *Y*. This is the shortest, and therefore the fastest, path through the circuit, because the input travels through only a single gate to the output.

The propagation delay of a combinational circuit is the sum of the propagation delays through each element on the critical path. The contamination delay is the sum of the contamination delays through each element on the short path. These delays are illustrated in Figure 2.69 and are described by the following equations:

$$t_{pd} = 2t_{pd\_\mathrm{AND}} + t_{pd\_\mathrm{OR}} \tag{2.8}$$

$$t_{cd} = t_{cd\_\mathrm{AND}} \tag{2.9}$$

Although we are ignoring wire delay in this analysis, digital circuits are now so fast that the delay of long wires can be as important as the delay of the gates. The speed of light delay in wires is covered in Appendix A.

Example 2.15 FINDING DELAYS

Ben Bitdiddle needs to find the propagation delay and contamination delay of the circuit shown in Figure 2.70. According to his data book, each gate has a propagation delay of 100 picoseconds (ps) and a contamination delay of 60 ps.

Solution: Ben begins by finding the critical path and the shortest path through the circuit. The critical path, highlighted in blue in Figure 2.71, is from input *A* or *B* through three gates to the output *Y*. Hence, t_{pd} is three times the propagation delay of a single gate, or 300 ps.

The shortest path, shown in gray in Figure 2.72, is from input *C*, *D*, or *E* through two gates to the output *Y*. There are only two gates in the shortest path, so t_{cd} is 120 ps.

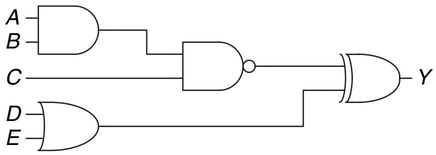


Figure 2.70 Ben's circuit

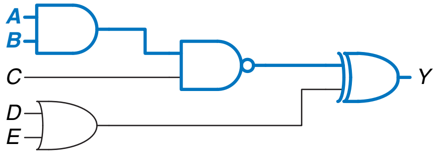


Figure 2.71 Ben's critical path

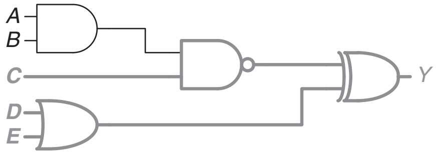


Figure 2.72 Ben's shortest path

Example 2.16 MULTIPLEXER TIMING: CONTROL-CRITICAL VS. DATA-CRITICAL

Compare the worst-case timing of the three four-input multiplexer designs shown in Figure 2.58 in Section 2.8.1. Table 2.7 lists the propagation delays for the components. What is the critical path for each design? Given your timing analysis, why might you choose one design over the other?

Solution: One of the critical paths for each of the three design options is highlighted in blue in Figures 2.73 and 2.74. t_{pd_sy} indicates the propagation delay from input *S* to output *Y*; t_{pd_dy} indicates the propagation delay from input *D* to output *Y*; t_{pd} is the worst of the two: $\max(t_{pd\_sy}, t_{pd\_dy})$.

For both the two-level logic and tristate implementations in Figure 2.73, the critical path is from one of the control signals *S* to the output *Y*: $t_{pd} = t_{pd\_sy}$. These circuits are *control critical*, because the critical path is from the control signals to the output. Any additional delay in the control signals will add directly to the worst-case delay. The delay from *D* to *Y* in Figure 2.73(b) is only 50 ps, compared with the delay from *S* to *Y* of 125 ps.

Figure 2.74 shows the hierarchical implementation of the 4:1 multiplexer using two stages of 2:1 multiplexers. The critical path is from any of the D inputs to the output. This circuit is *data critical*, because the critical path is from the data input to the output: $t_{pd} = t_{pd\_dy}$.

If data inputs arrive well before the control inputs, we would prefer the design with the shortest control-to-output delay (the hierarchical design in Figure 2.74). Similarly, if the control inputs arrive well before the data inputs, we would prefer the design with the shortest data-to-output delay (the tristate design in Figure 2.73(b)).

The best choice depends not only on the critical path through the circuit and the input arrival times, but also on the power, cost, and availability of parts.

Table 2.7 **Timing specifications for multiplexer circuit elements**

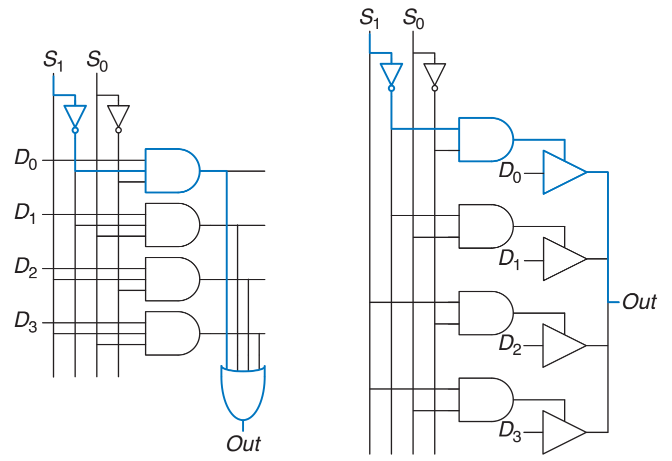
Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (A to Y)	50
tristate (enable to Y)	35

2.9.2 Glitches

So far we have discussed the case where a single input transition causes a single output transition. However, it is possible that a single input transition can cause *multiple* output transitions. These are called *glitches* or *hazards*. Although glitches usually don't cause problems, it is important to realize that they exist and recognize them when looking at timing diagrams. Figure 2.75 shows a circuit with a glitch and the Karnaugh map of the circuit.

The Boolean equation is correctly minimized, but let's look at what happens when $A = 0$, $C = 1$, and B transitions from 1 to 0. Figure 2.76 (see page 94) illustrates this scenario. The short path (shown in gray) goes through two gates, the AND and OR gates. The critical path (shown in blue) goes through an inverter and two gates, the AND and OR gates.

Hazards have another meaning related to microarchitecture in Chapter 7, so we will stick with the term *glitches* for multiple output transitions to avoid confusion.



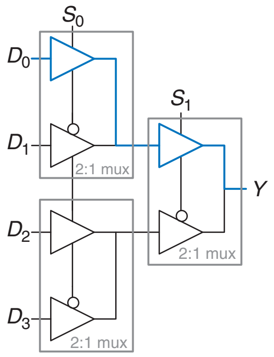
(a)

$$
\begin{aligned}
t_{pd\_sy} &= t_{pd\_\mathrm{INV}} + t_{pd\_\mathrm{AND3}} + t_{pd\_\mathrm{OR4}} \\
&= 30\text{ ps} + 80\text{ ps} + 90\text{ ps} \\
&= \mathbf{200\text{ ps}} \\
t_{pd\_dy} &= t_{pd\_\mathrm{AND3}} + t_{pd\_\mathrm{OR4}} \\
&= \mathbf{170\text{ ps}}
\end{aligned}
$$

(b)

$$
\begin{aligned}
t_{pd\_sy} &= t_{pd\_\mathrm{INV}} + t_{pd\_\mathrm{AND2}} + t_{pd\_\mathrm{TRI}\_sy} \\
&= 30\text{ ps} + 60\text{ ps} + 35\text{ ps} \\
&= \mathbf{125\text{ ps}} \\
t_{pd\_dy} &= t_{pd\_\mathrm{TRI}\_ay} \\
&= \mathbf{50\text{ ps}}
\end{aligned}
$$

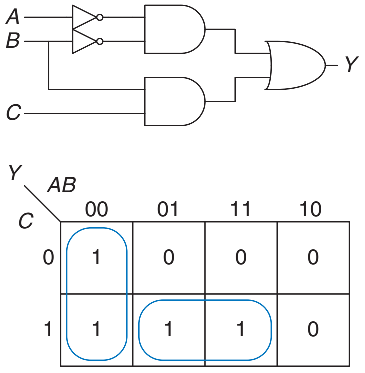
Figure 2.73 **4:1 multiplexer propagation delays: (a) two-level logic, (b) tristate**



$$t_{pd\_s0y} = t_{pd\_\mathrm{TRLSY}} + t_{pd\_\mathrm{TRI\_AY}} = \mathbf{85\text{ ns}}$$

$$t_{pd\_dy} = 2\; t_{pd\_\mathrm{TRI\_AY}} = \mathbf{100\text{ ns}}$$

Figure 2.74 **4:1 multiplexer propagation delays: hierarchical using 2:1 multiplexers**



Y / C \ AB	00	01	11	10
0	1	0	0	0
1	1	1	1	0

$$Y = \overline{A}\,\overline{B} + BC$$

Figure 2.75 **Circuit with a glitch**

As B transitions from 1 to 0, n2 (on the short path) falls before n1 (on the critical path) can rise. Until n1 rises, the two inputs to the OR gate are 0, and the output Y drops to 0. When n1 eventually rises, Y returns to 1. As shown in the timing diagram of Figure 2.76, Y starts at 1 and ends at 1 but momentarily glitches to 0.

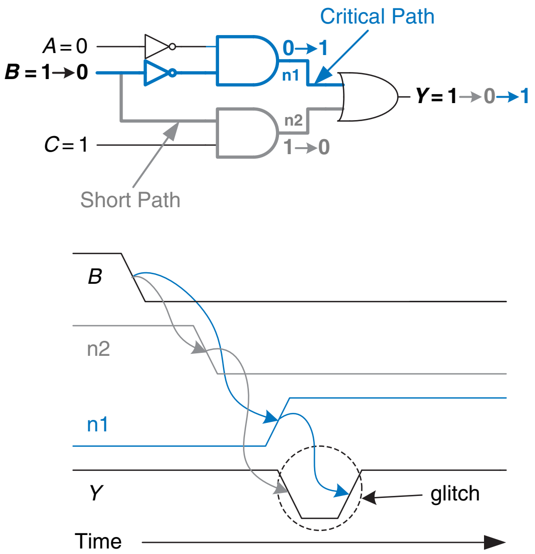

Critical Path
A = 0
B = 1→0
C = 1
0→1
n1
n2
1→0
Y = 1→0→1
Short Path
B
n2
n1
Y
glitch
Time

Figure 2.76 Timing of a glitch

As long as we wait for the propagation delay to elapse before we depend on the output, glitches are not a problem, because the output eventually settles to the right answer.

If we choose to, we can avoid this glitch by adding another gate to the implementation. This is easiest to understand in terms of the K-map. Figure 2.77 shows how an input transition on B from $ABC = 001$ to $ABC = 011$ moves from one prime implicant circle to another. The transition across the boundary of two prime implicants in the K-map indicates a possible glitch.

As we saw from the timing diagram in Figure 2.76, if the circuitry implementing one of the prime implicants turns *off* before the circuitry of the other prime implicant can turn *on*, there is a glitch. To fix this, we add another circle that *covers* that prime implicant boundary, as shown in Figure 2.78. You might recognize this as the consensus theorem, where the added term, $\overline{A}C$, is the consensus or redundant term.

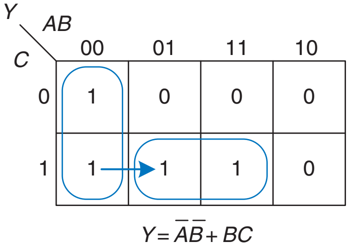

Y
AB
C
00
01
11
10
0
1
1
0
0
0
1
1
1
0
Y = $\overline{A}\,\overline{B}$ + BC

Figure 2.77 Input change crosses implicant boundary

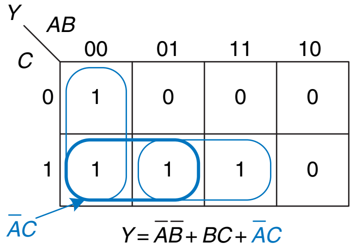


Figure 2.78 **K-map without glitch**

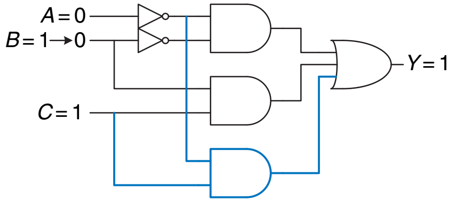


Figure 2.79 **Circuit without glitch**

Figure 2.79 shows the glitch-proof circuit. The added AND gate is highlighted in blue. Now a transition on B when $A = 0$ and $C = 1$ does not cause a glitch on the output, because the blue AND gate outputs 1 throughout the transition.

In general, a glitch can occur when a change in a single variable crosses the boundary between two prime implicants in a K-map. We can eliminate the glitch by adding redundant implicants to the K-map to cover these boundaries. This of course comes at the cost of extra hardware.

However, simultaneous transitions on multiple inputs can also cause glitches. These glitches cannot be fixed by adding hardware. Because the vast majority of interesting systems have simultaneous (or near-simultaneous) transitions on multiple inputs, glitches are a fact of life in most circuits. Although we have shown how to eliminate one kind of glitch, the point of discussing glitches is not to eliminate them but to be aware that they exist. This is especially important when looking at timing diagrams on a simulator or oscilloscope.

2.10 SUMMARY

A digital circuit is a module with discrete-valued inputs and outputs and a specification describing the function and timing of the module. This chapter has focused on combinational circuits, circuits whose outputs depend only on the current values of the inputs.

The function of a combinational circuit can be given by a truth table or a Boolean equation. The Boolean equation for any truth table can be obtained systematically using sum-of-products or product-of-sums form. In sum-of-products form, the function is written as the sum (OR) of one or more implicants. Implicants are the product (AND) of literals. Literals are the true or complementary forms of the input variables.

Boolean equations can be simplified using the rules of Boolean algebra. In particular, they can be simplified into minimal sum-of-products form by combining implicants that differ only in the true and complementary forms of one of the literals: $PA + P\overline{A} = P$. Karnaugh maps are a visual tool for minimizing functions of up to four variables. With practice, designers can usually simplify functions of a few variables by inspection. Computer-aided design tools are used for more complicated functions; such methods and tools are discussed in Chapter 4.

Logic gates are connected to create combinational circuits that perform the desired function. Any function in sum-of-products form can be built using two-level logic: NOT gates form the complements of the inputs, AND gates form the products, and OR gates form the sum. Depending on the function and the building blocks available, multilevel logic implementations with various types of gates may be more efficient. For example, CMOS circuits favor NAND and NOR gates because these gates can be built directly from CMOS transistors without requiring extra NOT gates. When using NAND and NOR gates, bubble pushing is helpful to keep track of the inversions.

Logic gates are combined to produce larger circuits such as multiplexers, decoders, and priority circuits. A multiplexer chooses one of the data inputs based on the select input. A decoder sets one of the outputs HIGH according to the inputs. A priority circuit produces an output indicating the highest priority input. These circuits are all examples of combinational building blocks. Chapter 5 will introduce more building blocks, including other arithmetic circuits. These building blocks will be used extensively to build a microprocessor in Chapter 7.

The timing specification of a combinational circuit consists of the propagation and contamination delays through the circuit. These indicate the longest and shortest times between an input change and the consequent output change. Calculating the propagation delay of a circuit involves identifying the critical path through the circuit, then adding up the propagation delays of each element along that path. There are many different ways to implement complicated combinational circuits; these ways offer trade-offs between speed and cost.

The next chapter will move to sequential circuits, whose outputs depend on current as well as previous values of the inputs. In other words, sequential circuits have *memory* of the past.

Exercises

Exercise 2.1 Write a Boolean equation in sum-of-products canonical form for each of the truth tables in Figure 2.80.

(a)

A	B	Y
0	0	1
0	1	0
1	0	1
1	1	1

(b)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(c)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(d)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

(e)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 2.80 Truth tables for Exercises 2.1 and 2.3

Exercise 2.2 Write a Boolean equation in sum-of-products canonical form for each of the truth tables in Figure 2.81.

(a)

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

(b)

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(c)

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(d)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

(e)

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Figure 2.81 Truth tables for Exercises 2.2 and 2.4

Exercise 2.3 Write a Boolean equation in product-of-sums canonical form for the truth tables in Figure 2.80.

Exercise 2.4 Write a Boolean equation in product-of-sums canonical form for the truth tables in Figure 2.81.

Exercise 2.5 Minimize each of the Boolean equations from Exercise 2.1.

Exercise 2.6 Minimize each of the Boolean equations from Exercise 2.2.

Exercise 2.7 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.5. Reasonably simple means that you are not wasteful of gates, but you don't waste vast amounts of time checking every possible implementation of the circuit either.

Exercise 2.8 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.6.

Exercise 2.9 Repeat Exercise 2.7 using only NOT gates and AND and OR gates.

Exercise 2.10 Repeat Exercise 2.8 using only NOT gates and AND and OR gates.

Exercise 2.11 Repeat Exercise 2.7 using only NOT gates and NAND and NOR gates.

Exercise 2.12 Repeat Exercise 2.8 using only NOT gates and NAND and NOR gates.

Exercise 2.13 Simplify the following Boolean equations using Boolean theorems. Check for correctness using a truth table or K-map.

(a) $Y = AC + \overline{A}\,\overline{B}C$

(b) $Y = \overline{A}\,\overline{B} + \overline{A}B\overline{C} + \overline{(A + \overline{C})}$

(c) $Y = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}\,\overline{C} + A\overline{B}C\overline{D} + ABD + \overline{A}\,\overline{B}C\overline{D} + B\overline{C}D + \overline{A}$

Exercise 2.14 Simplify the following Boolean equations using Boolean theorems. Check for correctness using a truth table or K-map.

(a) $Y = \overline{A}BC + \overline{A}B\overline{C}$

(b) $Y = \overline{ABC} + A\overline{B}$

(c) $Y = ABC\overline{D} + A\overline{BCD} + \overline{(A + B + C + D)}$

Exercise 2.15 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.13.

Exercise 2.16 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.14.

Exercise 2.17 Simplify each of the following Boolean equations. Sketch a reasonably simple combinational circuit implementing the simplified equation.

(a) $Y = BC + \overline{A}\,\overline{B}\,\overline{C} + B\overline{C}$

(b) $Y = \overline{A + \overline{A}B + \overline{A}\,\overline{B}} + \overline{A + \overline{B}}$

(c) $Y = ABC + ABD + ABE + ACD + ACE + (\overline{A + D + E}) + \overline{B}\,\overline{C}D$
$+ \overline{B}\,\overline{C}E + \overline{B}\,\overline{D}\,\overline{E} + \overline{C}\,\overline{D}\,\overline{E}$

Exercise 2.18 Simplify each of the following Boolean equations. Sketch a reasonably simple combinational circuit implementing the simplified equation.

(a) $Y = \overline{A}BC + \overline{B\overline{C}} + BC$

(b) $Y = (\overline{A + B + C})D + AD + B$

(c) $Y = ABCD + \overline{A}B\overline{C}D + \overline{(\overline{B} + D)}E$

Exercise 2.19 Give an example of a truth table requiring between 3 billion and 5 billion rows that can be constructed using fewer than 40 (but at least 1) two-input gates.

Exercise 2.20 Give an example of a circuit with a cyclic path that is nevertheless combinational.

Exercise 2.21 Alyssa P. Hacker says that any Boolean function can be written in minimal sum-of-products form as the sum of all of the prime implicants of the function. Ben Bitdiddle says that there are some functions whose minimal equation does not involve all of the prime implicants. Explain why Alyssa is right or provide a counterexample demonstrating Ben's point.

Exercise 2.22 Prove that the following theorems are true using perfect induction. You need not prove their duals.

(a) The idempotency theorem (T3)

(b) The distributivity theorem (T8)

(c) The combining theorem (T10)

Exercise 2.23 Prove De Morgan's Theorem (T12) for three variables, B_2, B_1, B_0, using perfect induction.

Exercise 2.24 Write Boolean equations for the circuit in Figure 2.82. You need not minimize the equations.

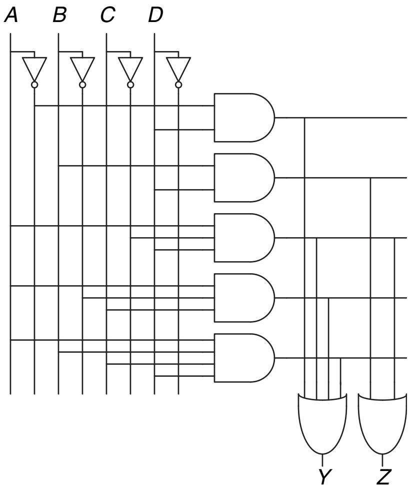


Figure 2.82 **Circuit schematic**

Exercise 2.25 Minimize the Boolean equations from Exercise 2.24 and sketch an improved circuit with the same function.

Exercise 2.26 Using De Morgan equivalent gates and bubble pushing methods, redraw the circuit in Figure 2.83 so that you can find the Boolean equation by inspection. Write the Boolean equation.

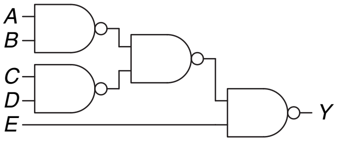


Figure 2.83 **Circuit schematic**

Exercise 2.27 Repeat Exercise 2.26 for the circuit in Figure 2.84.

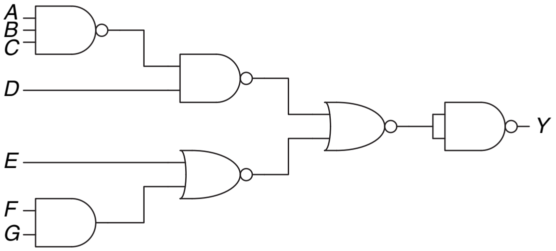


Figure 2.84 **Circuit schematic**

Exercise 2.28 Find a minimal Boolean equation for the function in Figure 2.85. Remember to take advantage of the don't care entries.

A	B	C	D	Y
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	0
0	1	0	0	0
0	1	0	1	X
0	1	1	0	0
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Figure 2.85 **Truth table for Exercise 2.28**

Exercise 2.29 Sketch a circuit for the function from Exercise 2.28.

Exercise 2.30 Does your circuit from Exercise 2.29 have any potential glitches when one of the inputs changes? If not, explain why not. If so, show how to modify the circuit to eliminate the glitches.

Exercise 2.31 Find a minimal Boolean equation for the function in Figure 2.86. Remember to take advantage of the don't care entries.

A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	1
0	0	1	0	X
0	0	1	1	X
0	1	0	0	0
0	1	0	1	X
0	1	1	0	X
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Figure 2.86 **Truth table for Exercise 2.31**

Exercise 2.32 Sketch a circuit for the function from Exercise 2.31.

Exercise 2.33 Ben Bitdiddle will enjoy his picnic on sunny days that have no ants. He will also enjoy his picnic any day he sees a hummingbird, as well as on days where there are ants and ladybugs. Write a Boolean equation for his enjoyment (E) in terms of sun (S), ants (A), hummingbirds (H), and ladybugs (L).

Exercise 2.34 Complete the design of the seven-segment decoder segments S_c through S_g (see Example 2.10):

(a) Derive Boolean equations for the outputs S_c through S_g assuming that inputs greater than 9 must produce blank (0) outputs.

(b) Derive Boolean equations for the outputs S_c through S_g assuming that inputs greater than 9 are don't cares.

(c) Sketch a reasonably simple gate-level implementation of part (b). Multiple outputs can share gates where appropriate.

Exercise 2.35 A circuit has four inputs and two outputs. The inputs $A_{3:0}$ represent a number from 0 to 15. Output P should be TRUE if the number is prime (0 and 1 are not prime, but 2, 3, 5, and so on, are prime). Output D should be TRUE if the number is divisible by 3. Give simplified Boolean equations for each output and sketch a circuit.

Exercise 2.36 A *priority encoder* has 2^N inputs. It produces an N-bit binary output indicating the most significant bit of the input that is TRUE, or 0 if none of the inputs are TRUE. It also produces an output $NONE$ that is TRUE if none of the inputs are TRUE. Design an eight-input priority encoder with inputs $A_{7:0}$ and outputs $Y_{2.0}$ and *NONE*. For example, if the input is 00100000, the output Y should be 101 and *NONE* should be 0. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.37 Design a modified priority encoder (see Exercise 2.36) that receives an 8-bit input, $A_{7:0}$, and produces two 3-bit outputs, $Y_{2:0}$ and $Z_{2:0}$ Y indicates the most significant bit of the input that is TRUE. Z indicates the second most significant bit of the input that is TRUE. Y should be 0 if none of the inputs are TRUE. Z should be 0 if no more than one of the inputs is TRUE. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.38 An M-bit *thermometer code* for the number k consists of k 1's in the least significant bit positions and $M - k$ 0's in all the more significant bit positions. A *binary-to-thermometer code converter* has N inputs and 2^N–1 outputs. It produces a 2^N–1 bit thermometer code for the number specified by the input. For example, if the input is 110, the output should be 0111111. Design a 3:7 binary-to-thermometer code converter. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.39 Write a minimized Boolean equation for the function performed by the circuit in Figure 2.87.

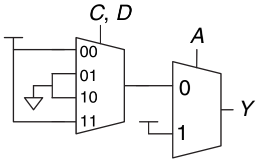


Figure 2.87 Multiplexer circuit

Exercise 2.40 Write a minimized Boolean equation for the function performed by the circuit in Figure 2.88.

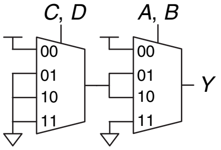


Figure 2.88 Multiplexer circuit

Exercise 2.41 Implement the function from Figure 2.80(b) using

(a) an 8:1 multiplexer

(b) a 4:1 multiplexer and one inverter

(c) a 2:1 multiplexer and two other logic gates

Exercise 2.42 Implement the function from Exercise 2.17(a) using

(a) an 8:1 multiplexer

(b) a 4:1 multiplexer and no other gates

(c) a 2:1 multiplexer, one OR gate, and an inverter

Exercise 2.43 Determine the propagation delay and contamination delay of the circuit in Figure 2.83. Use the gate delays given in Table 2.8.

Exercise 2.44 Determine the propagation delay and contamination delay of the circuit in Figure 2.84. Use the gate delays given in Table 2.8.

Table 2.8 **Gate delays for Exercises 2.43–2.47**

Gate	t_{pd} (ps)	t_{cd} (ps)
NOT	15	10
2-input NAND	20	15
3-input NAND	30	25
2-input NOR	30	25
3-input NOR	45	35
2-input AND	30	25
3-input AND	40	30
2-input OR	40	30
3-input OR	55	45
2-input XOR	60	40

Exercise 2.45 Sketch a schematic for a fast 3:8 decoder. Suppose gate delays are given in Table 2.8 (and only the gates in that table are available). Design your decoder to have the shortest possible critical path, and indicate what that path is. What are its propagation delay and contamination delay?

Exercise 2.46 Redesign the circuit from Exercise 2.35 to be as fast as possible. Use only the gates from Table 2.8. Sketch the new circuit and indicate the critical path. What are its propagation delay and contamination delay?

Exercise 2.47 Redesign the priority encoder from Exercise 2.36 to be as fast as possible. You may use any of the gates from Table 2.8. Sketch the new circuit and indicate the critical path. What are its propagation delay and contamination delay?

Exercise 2.48 Design an 8:1 multiplexer with the shortest possible delay from the data inputs to the output. You may use any of the gates from Table 2.7 on page 92. Sketch a schematic. Using the gate delays from the table, determine this delay.

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 2.1 Sketch a schematic for the two-input XOR function using only NAND gates. How few can you use?

Question 2.2 Design a circuit that will tell whether a given month has 31 days in it. The month is specified by a 4-bit input $A_{3:0}$. For example, if the inputs are 0001, the month is January, and if the inputs are 1100, the month is December. The circuit output Y should be HIGH only when the month specified by the inputs has 31 days in it. Write the simplified equation, and draw the circuit diagram using a minimum number of gates. (Hint: Remember to take advantage of don't cares.)

Question 2.3 What is a tristate buffer? How and why is it used?

Question 2.4 A gate or set of gates is universal if it can be used to construct any Boolean function. For example, the set {AND, OR, NOT} is universal.

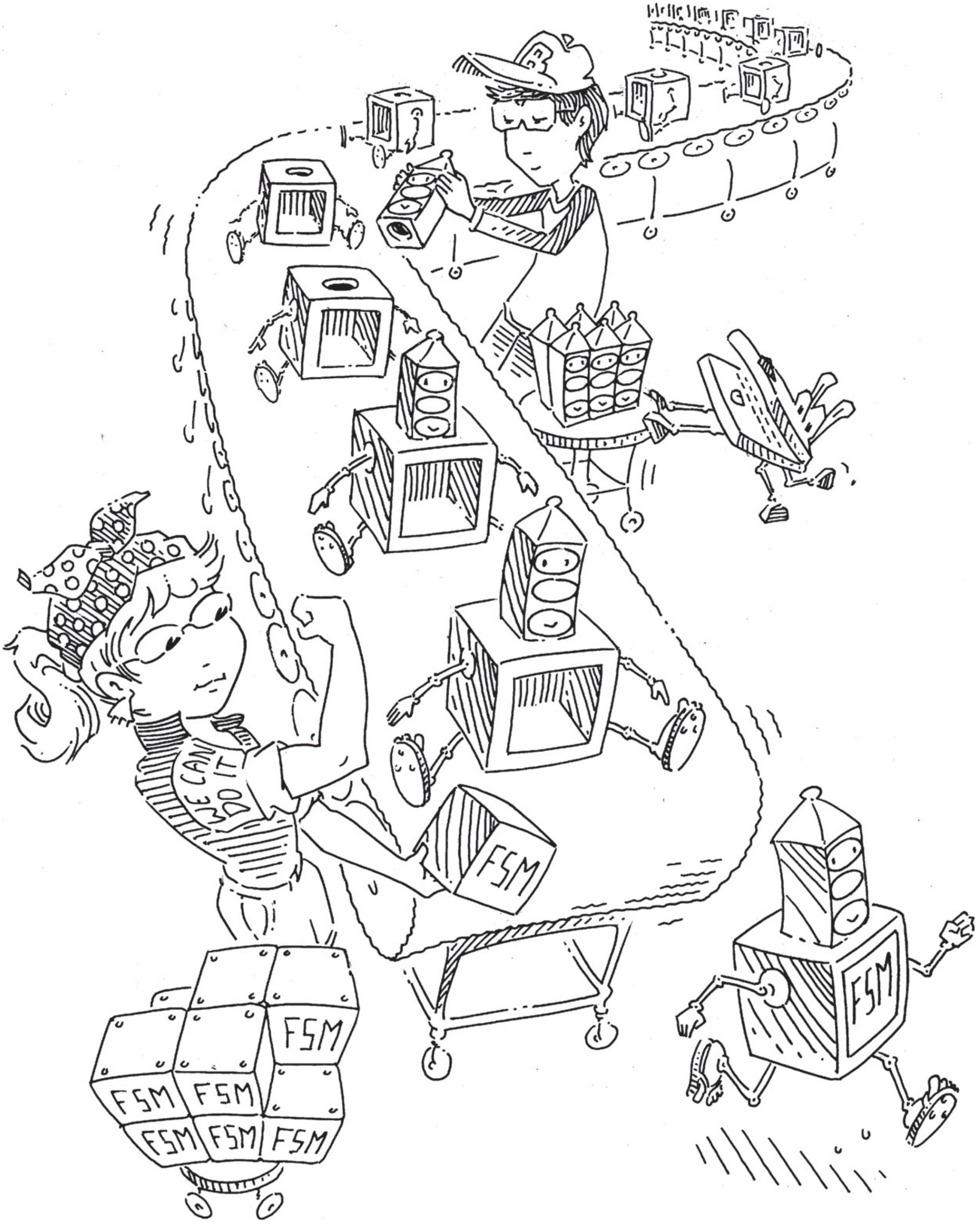
(a) Is an AND gate by itself universal? Why or why not?

(b) Is the set {OR, NOT} universal? Why or why not?

(c) Is a NAND gate by itself universal? Why or why not?

Question 2.5 Explain why a circuit's contamination delay might be less than (instead of equal to) its propagation delay.

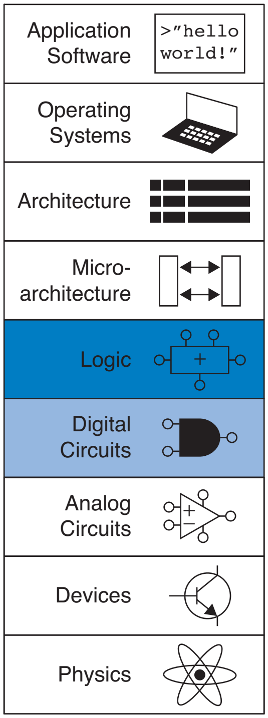
This page intentionally left blank


WE CAN DO IT
FSM
FSM
FSM
FSM
FSM
FSM
FSM
FSM

3

Sequential Logic Design

3.1 Introduction
3.2 Latches and Flip-Flops
3.3 Synchronous Logic Design
3.4 Finite State Machines
3.5 Timing of Sequential Logic
3.6 Parallelism
3.7 Summary
Exercises
Interview Questions



3.1 INTRODUCTION

In the last chapter, we showed how to analyze and design combinational logic. The output of combinational logic depends only on current input values. Given a specification in the form of a truth table or Boolean equation, we can create an optimized circuit to meet the specification.

In this chapter, we will analyze and design *sequential* logic. The outputs of sequential logic depend on both current and prior input values. Hence, sequential logic has memory. Sequential logic might explicitly remember certain previous inputs, or it might distill the prior inputs into a smaller amount of information called the *state* of the system. The state of a digital sequential circuit is a set of bits called *state variables* that contain all the information about the past necessary to explain the future behavior of the circuit.

The chapter begins by studying latches and flip-flops, which are simple sequential circuits that store one bit of state. In general, sequential circuits are complicated to analyze. To simplify design, we discipline ourselves to build only synchronous sequential circuits consisting of combinational logic and banks of flip-flops containing the state of the circuit. The chapter describes finite state machines, which are an easy way to design sequential circuits. Finally, we analyze the speed of sequential circuits and discuss parallelism as a way to increase speed.

3.2 LATCHES AND FLIP-FLOPS

The fundamental building block of memory is a *bistable* element, an element with two stable states. Figure 3.1(a) shows a simple bistable element consisting of a pair of inverters connected in a loop. Figure 3.1(b) shows the same circuit redrawn to emphasize the symmetry. The inverters are *cross-coupled*, meaning that the input of I1 is the output of I2 and vice versa. The circuit has no inputs, but it does have two outputs, Q and $\overline{Q}$.

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00003-3
© 2013 Elsevier, Inc. All rights reserved.

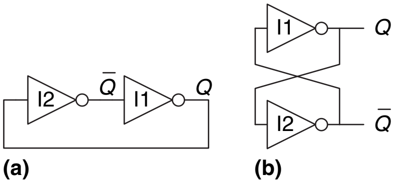


Figure 3.1 Cross-coupled inverter pair

Just as Y is commonly used for the output of combinational logic, Q is commonly used for the output of sequential logic.

Analyzing this circuit is different from analyzing a combinational circuit because it is cyclic: Q depends on $\overline{Q}$, and $\overline{Q}$ depends on Q.

Consider the two cases, Q is 0 or Q is 1. Working through the consequences of each case, we have:

- *Case I:* $Q = 0$
  As shown in Figure 3.2(a), I2 receives a FALSE input, Q, so it produces a TRUE output on $\overline{Q}$. I1 receives a TRUE input, $\overline{Q}$, so it produces a FALSE output on Q. This is consistent with the original assumption that $Q = 0$, so the case is said to be *stable*.

- *Case II:* $Q = 1$
  As shown in Figure 3.2(b), I2 receives a TRUE input and produces a FALSE output on $\overline{Q}$. I1 receives a FALSE input and produces a TRUE output on Q. This is again stable.

Because the cross-coupled inverters have two stable states, $Q = 0$ and $Q = 1$, the circuit is said to be bistable. A subtle point is that the circuit has a third possible state with both outputs approximately halfway between 0 and 1. This is called a *metastable* state and will be discussed in Section 3.5.4.

An element with N stable states conveys $\log_2 N$ bits of information, so a bistable element stores one bit. The state of the cross-coupled inverters is contained in one binary state variable, Q. The value of Q tells us everything about the past that is necessary to explain the future behavior of the circuit. Specifically, if $Q = 0$, it will remain 0 forever, and if $Q = 1$, it will remain 1 forever. The circuit does have another node, $\overline{Q}$, but $\overline{Q}$ does not contain any additional information because if Q is known, $\overline{Q}$ is also known. On the other hand, $\overline{Q}$ is also an acceptable choice for the state variable.

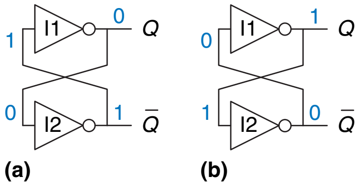


Figure 3.2 Bistable operation of cross-coupled inverters

When power is first applied to a sequential circuit, the initial state is unknown and usually unpredictable. It may differ each time the circuit is turned on.

Although the cross-coupled inverters can store a bit of information, they are not practical because the user has no inputs to control the state. However, other bistable elements, such as *latches* and *flip-flops*, provide inputs to control the value of the state variable. The remainder of this section considers these circuits.

3.2.1 SR Latch

One of the simplest sequential circuits is the *SR latch*, which is composed of two cross-coupled NOR gates, as shown in Figure 3.3. The latch has two inputs, S and R, and two outputs, Q and $\overline{Q}$. The SR latch is similar to the cross-coupled inverters, but its state can be controlled through the S and R inputs, which *set* and *reset* the output Q.

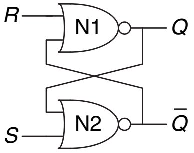


Figure 3.3 **SR latch schematic**

A good way to understand an unfamiliar circuit is to work out its truth table, so that is where we begin. Recall that a NOR gate produces a FALSE output when either input is TRUE. Consider the four possible combinations of R and S.

- *Case I:* $R = 1, S = 0$
  N1 sees at least one TRUE input, R, so it produces a FALSE output on Q. N2 sees both Q and S FALSE, so it produces a TRUE output on $\overline{Q}$.
- *Case II:* $R = 0, S = 1$
  N1 receives inputs of 0 and $\overline{Q}$. Because we don't yet know $\overline{Q}$, we can't determine the output Q. N2 receives at least one TRUE input, S, so it produces a FALSE output on $\overline{Q}$. Now we can revisit N1, knowing that both inputs are FALSE, so the output Q is TRUE.
- *Case III:* $R = 1, S = 1$
  N1 and N2 both see at least one TRUE input (R or S), so each produces a FALSE output. Hence Q and $\overline{Q}$ are both FALSE.
- *Case IV:* $R = 0, S = 0$
  N1 receives inputs of 0 and $\overline{Q}$. Because we don't yet know $\overline{Q}$, we can't determine the output. N2 receives inputs of 0 and Q. Because we don't yet know Q, we can't determine the output. Now we are stuck. This is reminiscent of the cross-coupled inverters. But we know that Q must either be 0 or 1. So we can solve the problem by checking what happens in each of these subcases.

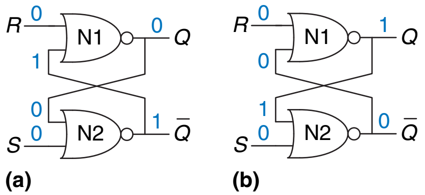


Figure 3.4 **Bistable states of SR latch**

- *Case IVa:* $Q = 0$
  Because S and Q are FALSE, N2 produces a TRUE output on $\overline{Q}$, as shown in Figure 3.4(a). Now N1 receives one TRUE input, $\overline{Q}$, so its output, Q, is FALSE, just as we had assumed.
- *Case IVb:* $Q = 1$
  Because Q is TRUE, N2 produces a FALSE output on $\overline{Q}$, as shown in Figure 3.4(b). Now N1 receives two FALSE inputs, R and $\overline{Q}$, so its output, Q, is TRUE, just as we had assumed.

Putting this all together, suppose Q has some known prior value, which we will call Q_{prev}, before we enter Case IV. Q_{prev} is either 0 or 1, and represents the state of the system. When R and S are 0, Q will remember this old value, Q_{prev}, and $\overline{Q}$ will be its complement, $\overline{Q}_{prev}$. This circuit has memory.

Case	S	R	Q	$\overline{Q}$
IV	0	0	Q_{prev}	$\overline{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Figure 3.5 **SR latch truth table**

The truth table in Figure 3.5 summarizes these four cases. The inputs S and R stand for *Set* and *Reset*. To *set* a bit means to make it TRUE. To *reset* a bit means to make it FALSE. The outputs, Q and $\overline{Q}$, are normally complementary. When R is asserted, Q is reset to 0 and $\overline{Q}$ does the opposite. When S is asserted, Q is set to 1 and $\overline{Q}$ does the opposite. When neither input is asserted, Q remembers its old value, Q_{prev}. Asserting both S and R simultaneously doesn't make much sense because it means the latch should be set and reset at the same time, which is impossible. The poor confused circuit responds by making both outputs 0.

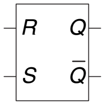


Figure 3.6 **SR latch symbol**

The SR latch is represented by the symbol in Figure 3.6. Using the symbol is an application of abstraction and modularity. There are various ways to build an SR latch, such as using different logic gates or transistors. Nevertheless, any circuit element with the relationship specified by the truth table in Figure 3.5 and the symbol in Figure 3.6 is called an SR latch.

Like the cross-coupled inverters, the SR latch is a bistable element with one bit of state stored in Q. However, the state can be controlled through the S and R inputs. When R is asserted, the state is reset to 0. When S is asserted, the state is set to 1. When neither is asserted, the state retains its old value. Notice that the entire history of inputs can be accounted for by the single state variable Q. No matter what pattern of setting and resetting occurred in the past, all that is needed to predict the future behavior of the SR latch is whether it was most recently set or reset.

3.2.2 D Latch

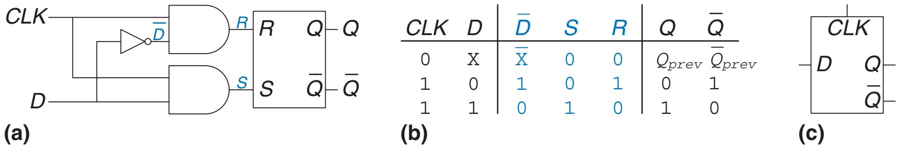
The SR latch is awkward because it behaves strangely when both S and R are simultaneously asserted. Moreover, the S and R inputs conflate the issues of *what* and *when*. Asserting one of the inputs determines not only *what* the state should be but also *when* it should change. Designing circuits becomes easier when these questions of what and when are separated. The D latch in Figure 3.7(a) solves these problems. It has two inputs. The *data* input, D, controls what the next state should be. The *clock* input, CLK, controls when the state should change.

Again, we analyze the latch by writing the truth table, given in Figure 3.7(b). For convenience, we first consider the internal nodes $\overline{D}$, S, and R. If $CLK = 0$, both S and R are FALSE, regardless of the value of D. If $CLK = 1$, one AND gate will produce TRUE and the other FALSE, depending on the value of D. Given S and R, Q and $\overline{Q}$ are determined using Figure 3.5. Observe that when $CLK = 0$, Q remembers its old value, Q_{prev}. When $CLK = 1$, $Q = D$. In all cases, $\overline{Q}$ is the complement of Q, as would seem logical. The D latch avoids the strange case of simultaneously asserted R and S inputs.

Putting it all together, we see that the clock controls when data flows through the latch. When $CLK = 1$, the latch is *transparent*. The data at D flows through to Q as if the latch were just a buffer. When $CLK = 0$, the latch is *opaque*. It blocks the new data from flowing through to Q, and Q retains the old value. Hence, the D latch is sometimes called a *transparent latch* or a *level-sensitive* latch. The D latch symbol is given in Figure 3.7(c).

Some people call a latch open or closed rather than transparent or opaque. However, we think those terms are ambiguous—does *open* mean transparent like an open door, or opaque, like an open circuit?

The D latch updates its state continuously while $CLK = 1$. We shall see later in this chapter that it is useful to update the state only at a specific instant in time. The D flip-flop described in the next section does just that.



CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

Figure 3.7 D latch: (a) schematic, (b) truth table, (c) symbol

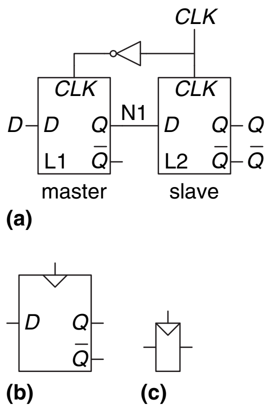


Figure 3.8 **D flip-flop: (a) schematic, (b) symbol, (c) condensed symbol**

3.2.3 D Flip-Flop

A *D flip-flop* can be built from two back-to-back D latches controlled by complementary clocks, as shown in Figure 3.8(a). The first latch, L1, is called the *master*. The second latch, L2, is called the *slave*. The node between them is named N1. A symbol for the D flip-flop is given in Figure 3.8(b). When the $\overline{Q}$ output is not needed, the symbol is often condensed as in Figure 3.8(c).

When $CLK = 0$, the master latch is transparent and the slave is opaque. Therefore, whatever value was at D propagates through to N1. When $CLK = 1$, the master goes opaque and the slave becomes transparent. The value at N1 propagates through to Q, but N1 is cut off from D. Hence, whatever value was at D immediately before the clock rises from 0 to 1 gets copied to Q immediately after the clock rises. At all other times, Q retains its old value, because there is always an opaque latch blocking the path between D and Q.

In other words, *a D flip-flop copies D to Q on the rising edge of the clock, and remembers its state at all other times*. Reread this definition until you have it memorized; one of the most common problems for beginning digital designers is to forget what a flip-flop does. The rising edge of the clock is often just called the *clock edge* for brevity. The D input specifies what the new state will be. The clock edge indicates when the state should be updated.

A D flip-flop is also known as a *master-slave flip-flop*, an *edge-triggered flip-flop*, or a *positive edge-triggered flip-flop*. The triangle in the symbols denotes an edge-triggered clock input. The $\overline{Q}$ output is often omitted when it is not needed.

The precise distinction between *flip-flops* and *latches* is somewhat muddled and has evolved over time. In common industry usage, a flip-flop is *edge-triggered*. In other words, it is a bistable element with a *clock* input. The state of the flip-flop changes only in response to a clock edge, such as when the clock rises from 0 to 1. Bistable elements without an edge-triggered clock are commonly called latches.

The term flip-flop or latch by itself usually refers to a *D flip-flop* or *D latch*, respectively, because these are the types most commonly used in practice.

Example 3.1 FLIP-FLOP TRANSISTOR COUNT

How many transistors are needed to build the D flip-flop described in this section?

Solution: A NAND or NOR gate uses four transistors. A NOT gate uses two transistors. An AND gate is built from a NAND and a NOT, so it uses six transistors. The SR latch uses two NOR gates, or eight transistors. The D latch uses an SR latch, two AND gates, and a NOT gate, or 22 transistors. The D flip-flop uses two D latches and a NOT gate, or 46 transistors. Section 3.2.7 describes a more efficient CMOS implementation using transmission gates.

3.2.4 Register

An N-bit register is a bank of N flip-flops that share a common CLK input, so that all bits of the register are updated at the same time. Registers are the key building block of most sequential circuits. Figure 3.9

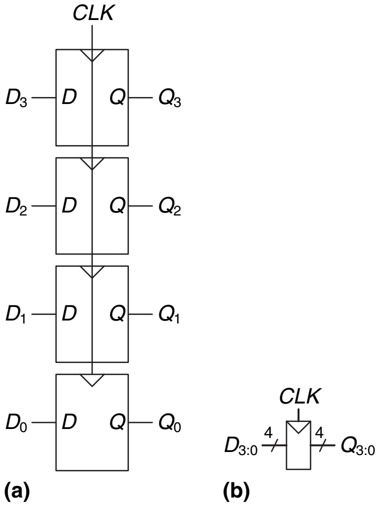


Figure 3.9 **A 4-bit register: (a) schematic and (b) symbol**

shows the schematic and symbol for a four-bit register with inputs $D_{3:0}$ and outputs $Q_{3:0}$. $D_{3:0}$ and $Q_{3:0}$ are both 4-bit busses.

3.2.5 Enabled Flip-Flop

An *enabled flip-flop* adds another input called *EN* or *ENABLE* to determine whether data is loaded on the clock edge. When *EN* is TRUE, the enabled flip-flop behaves like an ordinary D flip-flop. When *EN* is FALSE, the enabled flip-flop ignores the clock and retains its state. Enabled flip-flops are useful when we wish to load a new value into a flip-flop only some of the time, rather than on every clock edge.

Figure 3.10 shows two ways to construct an enabled flip-flop from a D flip-flop and an extra gate. In Figure 3.10(a), an input multiplexer chooses whether to pass the value at *D*, if *EN* is TRUE, or to recycle the old state from *Q*, if *EN* is FALSE. In Figure 3.10(b), the clock is *gated*. If *EN* is TRUE, the *CLK* input to the flip-flop toggles normally. If *EN* is

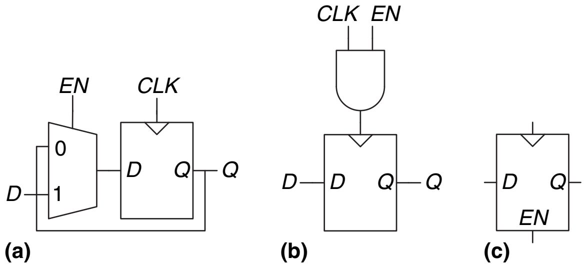
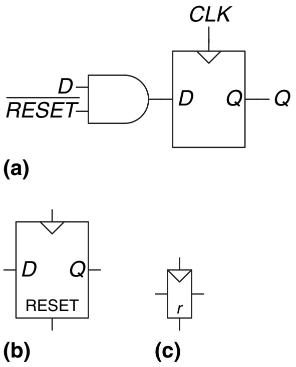


Figure 3.10 **Enabled flip-flop: (a, b) schematics, (c) symbol**

FALSE, the *CLK* input is also FALSE and the flip-flop retains its old value. Notice that *EN* must not change while $CLK = 1$, lest the flip-flop see a clock *glitch* (switch at an incorrect time). Generally, performing logic on the clock is a bad idea. Clock gating delays the clock and can cause timing errors, as we will see in Section 3.5.3, so do it only if you are sure you know what you are doing. The symbol for an enabled flip-flop is given in Figure 3.10(c).

3.2.6 Resettable Flip-Flop

A *resettable flip-flop* adds another input called *RESET*. When *RESET* is FALSE, the resettable flip-flop behaves like an ordinary D flip-flop. When *RESET* is TRUE, the resettable flip-flop ignores *D* and resets the output to 0. Resettable flip-flops are useful when we want to force a known state (i.e., 0) into all the flip-flops in a system when we first turn it on.



(a)

(b) (c)

Figure 3.11 **Synchronously resettable flip-flop: (a) schematic, (b, c) symbols**

Such flip-flops may be *synchronously* or *asynchronously resettable*. Synchronously resettable flip-flops reset themselves only on the rising edge of *CLK*. Asynchronously resettable flip-flops reset themselves as soon as *RESET* becomes TRUE, independent of *CLK*.

Figure 3.11(a) shows how to construct a synchronously resettable flip-flop from an ordinary D flip-flop and an AND gate. When $\overline{RESET}$ is FALSE, the AND gate forces a 0 into the input of the flip-flop. When $\overline{RESET}$ is TRUE, the AND gate passes *D* to the flip-flop. In this example, $\overline{RESET}$ is an *active low* signal, meaning that the reset signal performs its function when it is 0, not 1. By adding an inverter, the circuit could have accepted an active high reset signal instead. Figures 3.11(b) and 3.11(c) show symbols for the resettable flip-flop with active high reset.

Asynchronously resettable flip-flops require modifying the internal structure of the flip-flop and are left to you to design in Exercise 3.13; however, they are frequently available to the designer as a standard component.

As you might imagine, settable flip-flops are also occasionally used. They load a 1 into the flip-flop when SET is asserted, and they too come in synchronous and asynchronous flavors. Resettable and settable flip-flops may also have an enable input and may be grouped into *N*-bit registers.

3.2.7 Transistor-Level Latch and Flip-Flop Designs*

Example 3.1 showed that latches and flip-flops require a large number of transistors when built from logic gates. But the fundamental role of a latch is to be transparent or opaque, much like a switch. Recall from

Section 1.7.7 that a transmission gate is an efficient way to build a CMOS switch, so we might expect that we could take advantage of transmission gates to reduce the transistor count.

A compact D latch can be constructed from a single transmission gate, as shown in Figure 3.12(a). When $CLK = 1$ and $\overline{CLK} = 0$, the transmission gate is ON, so D flows to Q and the latch is transparent. When $CLK = 0$ and $\overline{CLK} = 1$, the transmission gate is OFF, so Q is isolated from D and the latch is opaque. This latch suffers from two major limitations:

- *Floating output node:* When the latch is opaque, Q is not held at its value by any gates. Thus Q is called a *floating* or *dynamic* node. After some time, noise and charge leakage may disturb the value of Q.
- *No buffers:* The lack of buffers has caused malfunctions on several commercial chips. A spike of noise that pulls D to a negative voltage can turn on the nMOS transistor, making the latch transparent, even when $CLK = 0$. Likewise, a spike on D above V_{DD} can turn on the pMOS transistor even when $CLK = 0$. And the transmission gate is symmetric, so it could be driven backward with noise on Q affecting the input D. The general rule is that neither the input of a transmission gate nor the state node of a sequential circuit should ever be exposed to the outside world, where noise is likely.

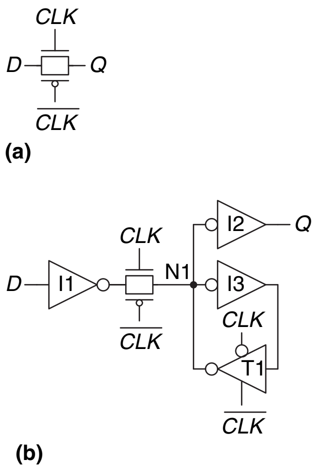


Figure 3.12 **D latch schematic**

Figure 3.12(b) shows a more robust 12-transistor D latch used on modern commercial chips. It is still built around a clocked transmission gate, but it adds inverters I1 and I2 to buffer the input and output. The state of the latch is held on node N1. Inverter I3 and the tristate buffer, T1, provide feedback to turn N1 into a *static node*. If a small amount of noise occurs on N1 while $CLK = 0$, T1 will drive N1 back to a valid logic value.

Figure 3.13 shows a D flip-flop constructed from two static latches controlled by $\overline{CLK}$ and CLK. Some redundant internal inverters have been removed, so the flip-flop requires only 20 transistors.

This circuit assumes CLK and $\overline{CLK}$ are both available. If not, two more transistors are needed for a CLK inverter.

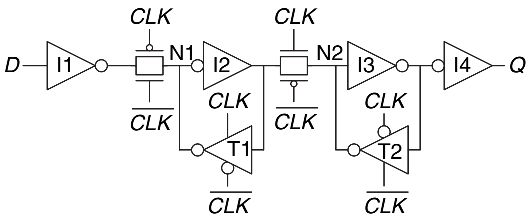


Figure 3.13 **D flip-flop schematic**

3.2.8 Putting It All Together

Latches and flip-flops are the fundamental building blocks of sequential circuits. Remember that a D latch is level-sensitive, whereas a D flip-flop is edge-triggered. The D latch is transparent when $CLK = 1$, allowing the input D to flow through to the output Q. The D flip-flop copies D to Q on the rising edge of CLK. At all other times, latches and flip-flops retain their old state. A register is a bank of several D flip-flops that share a common CLK signal.

Example 3.2 FLIP-FLOP AND LATCH COMPARISON

Ben Bitdiddle applies the D and CLK inputs shown in Figure 3.14 to a D latch and a D flip-flop. Help him determine the output, Q, of each device.

Solution: Figure 3.15 shows the output waveforms, assuming a small delay for Q to respond to input changes. The arrows indicate the cause of an output change. The initial value of Q is unknown and could be 0 or 1, as indicated by the pair of horizontal lines. First consider the latch. On the first rising edge of CLK, $D = 0$, so Q definitely becomes 0. Each time D changes while $CLK = 1$, Q also follows. When D changes while $CLK = 0$, it is ignored. Now consider the flip-flop. On each rising edge of CLK, D is copied to Q. At all other times, Q retains its state.

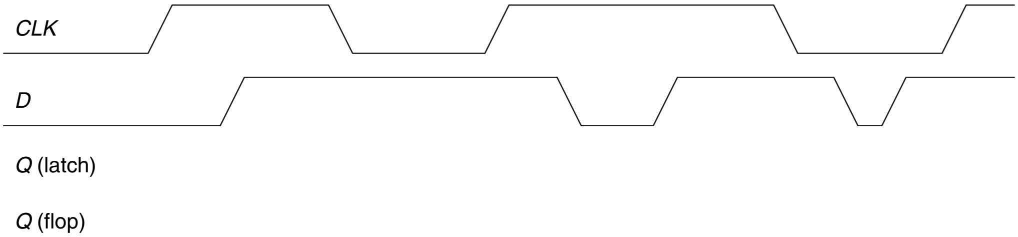


Figure 3.14 Example waveforms

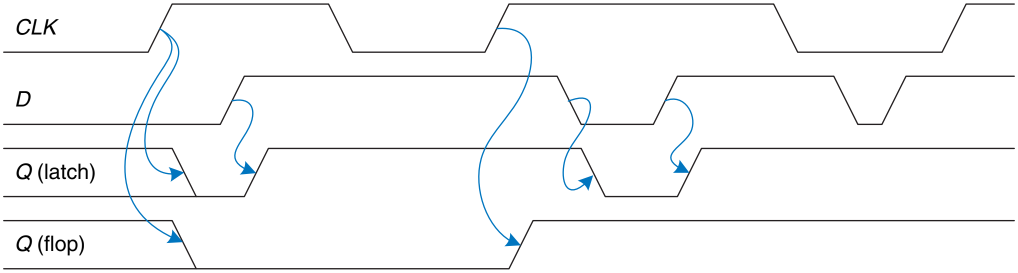


Figure 3.15 Solution waveforms

3.3 SYNCHRONOUS LOGIC DESIGN

In general, sequential circuits include all circuits that are not combinational—that is, those whose output cannot be determined simply by looking at the current inputs. Some sequential circuits are just plain kooky. This section begins by examining some of those curious circuits. It then introduces the notion of synchronous sequential circuits and the dynamic discipline. By disciplining ourselves to synchronous sequential circuits, we can develop easy, systematic ways to analyze and design sequential systems.

3.3.1 Some Problematic Circuits

Example 3.3 ASTABLE CIRCUITS

Alyssa P. Hacker encounters three misbegotten inverters who have tied themselves in a loop, as shown in Figure 3.16. The output of the third inverter is *fed back* to the first inverter. Each inverter has a propagation delay of 1 ns. Determine what the circuit does.

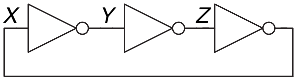


Figure 3.16 Three-inverter loop

Solution: Suppose node X is initially 0. Then $Y = 1$, $Z = 0$, and hence $X = 1$, which is inconsistent with our original assumption. The circuit has no stable states and is said to be *unstable* or *astable*. Figure 3.17 shows the behavior of the circuit. If X rises at time 0, Y will fall at 1 ns, Z will rise at 2 ns, and X will fall again at 3 ns. In turn, Y will rise at 4 ns, Z will fall at 5 ns, and X will rise again at 6 ns, and then the pattern will repeat. Each node oscillates between 0 and 1 with a *period* (repetition time) of 6 ns. This circuit is called a *ring oscillator*.

The period of the ring oscillator depends on the propagation delay of each inverter. This delay depends on how the inverter was manufactured, the power supply voltage, and even the temperature. Therefore, the ring oscillator period is difficult to accurately predict. In short, the ring oscillator is a sequential circuit with zero inputs and one output that changes periodically.

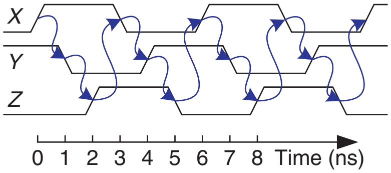


Figure 3.17 Ring oscillator waveforms

Example 3.4 RACE CONDITIONS

Ben Bitdiddle designed a new D latch that he claims is better than the one in Figure 3.7 because it uses fewer gates. He has written the truth table to find the

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$$

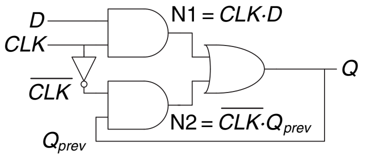


Figure 3.18 **An improved (?) D latch**

output, Q, given the two inputs, D and CLK, and the old state of the latch, Q_{prev}. Based on this truth table, he has derived Boolean equations. He obtains Q_{prev} by feeding back the output, Q. His design is shown in Figure 3.18. Does his latch work correctly, independent of the delays of each gate?

Solution: Figure 3.19 shows that the circuit has a *race condition* that causes it to fail when certain gates are slower than others. Suppose $CLK = D = 1$. The latch is transparent and passes D through to make $Q = 1$. Now, CLK falls. The latch should remember its old value, keeping $Q = 1$. However, suppose the delay through the inverter from CLK to $\overline{CLK}$ is rather long compared to the delays of the AND and OR gates. Then nodes N1 and Q may both fall before $\overline{CLK}$ rises. In such a case, N2 will never rise, and Q becomes stuck at 0.

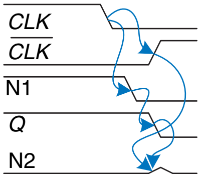


Figure 3.19 **Latch waveforms illustrating race condition**

This is an example of *asynchronous* circuit design in which outputs are directly fed back to inputs. Asynchronous circuits are infamous for having race conditions where the behavior of the circuit depends on which of two paths through logic gates is fastest. One circuit may work, while a seemingly identical one built from gates with slightly different delays may not work. Or the circuit may work only at certain temperatures or voltages at which the delays are just right. These malfunctions are extremely difficult to track down.

3.3.2 Synchronous Sequential Circuits

The previous two examples contain loops called *cyclic paths*, in which outputs are fed directly back to inputs. They are sequential rather than combinational circuits. Combinational logic has no cyclic paths and no races. If inputs are applied to combinational logic, the outputs will always settle to the correct value within a propagation delay. However, sequential circuits with cyclic paths can have undesirable races or unstable behavior. Analyzing such circuits for problems is time-consuming, and many bright people have made mistakes.

To avoid these problems, designers break the cyclic paths by inserting registers somewhere in the path. This transforms the circuit into a collection of combinational logic and registers. The registers contain the state of the system, which changes only at the clock edge, so we say the state is *synchronized* to the clock. If the clock is sufficiently slow, so that the inputs to all registers settle before the next clock edge, all races are eliminated. Adopting this discipline of always using registers in the feedback path leads us to the formal definition of a synchronous sequential circuit.

Recall that a circuit is defined by its input and output terminals and its functional and timing specifications. A sequential circuit has a finite set of discrete *states* $\{S_0, S_1, \ldots, S_{k-1}\}$. A *synchronous sequential circuit* has a clock input, whose rising edges indicate a sequence of times at which state transitions occur. We often use the terms *current state* and *next state* to distinguish the state of the system at the present from the state to which it will enter on the next clock edge. The functional specification details the next state and the value of each output for each possible combination of current state and input values. The timing specification consists of an upper bound, t_{pcq}, and a lower bound, t_{ccq}, on the time from the rising edge of the clock until the *output* changes, as well as *setup* and *hold* times, t_{setup} and t_{hold}, that indicate when the *inputs* must be stable relative to the rising edge of the clock.

The rules of *synchronous sequential circuit composition* teach us that a circuit is a synchronous sequential circuit if it consists of interconnected circuit elements such that

- Every circuit element is either a register or a combinational circuit
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register.

Sequential circuits that are not synchronous are called *asynchronous*.

A flip-flop is the simplest synchronous sequential circuit. It has one input, *D*, one clock, *CLK*, one output, *Q*, and two states, {0, 1}. The functional specification for a flip-flop is that the next state is *D* and that the output, *Q*, is the current state, as shown in Figure 3.20.

We often call the current state variable *S* and the next state variable *S*′. In this case, the prime after *S* indicates next state, not inversion. The timing of sequential circuits will be analyzed in Section 3.5.

Two other common types of synchronous sequential circuits are called finite state machines and pipelines. These will be covered later in this chapter.

t_{pcq} stands for the time of propagation from clock to *Q*, where *Q* indicates the output of a synchronous sequential circuit. t_{ccq} stands for the time of contamination from clock to *Q*. These are analogous to t_{pd} and t_{cd} in combinational logic.

This definition of a synchronous sequential circuit is sufficient, but more restrictive than necessary. For example, in high-performance microprocessors, some registers may receive delayed or gated clocks to squeeze out the last bit of performance or power. Similarly, some microprocessors use latches instead of registers. However, the definition is adequate for all of the synchronous sequential circuits covered in this book and for most commercial digital systems.

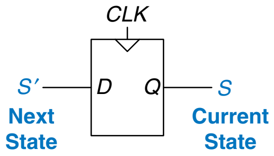


Figure 3.20 **Flip-flop current state and next state**

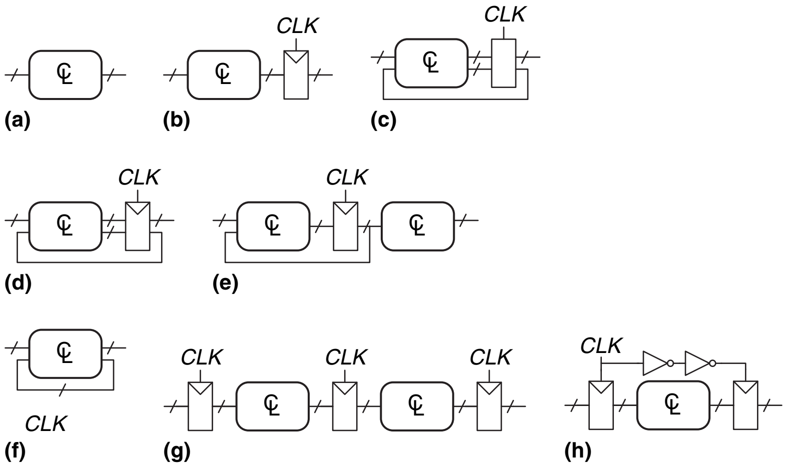


Figure 3.21 **Example circuits**

Example 3.5 SYNCHRONOUS SEQUENTIAL CIRCUITS

Which of the circuits in Figure 3.21 are synchronous sequential circuits?

Solution: Circuit (a) is combinational, not sequential, because it has no registers. (b) is a simple sequential circuit with no feedback. (c) is neither a combinational circuit nor a synchronous sequential circuit, because it has a latch that is neither a register nor a combinational circuit. (d) and (e) are synchronous sequential logic; they are two forms of finite state machines, which are discussed in Section 3.4. (f) is neither combinational nor synchronous sequential, because it has a cyclic path from the output of the combinational logic back to the input of the same logic but no register in the path. (g) is synchronous sequential logic in the form of a pipeline, which we will study in Section 3.6. (h) is not, strictly speaking, a synchronous sequential circuit, because the second register receives a different clock signal than the first, delayed by two inverter delays.

3.3.3 Synchronous and Asynchronous Circuits

Asynchronous design in theory is more general than synchronous design, because the timing of the system is not limited by clocked registers. Just as analog circuits are more general than digital circuits because analog circuits can use any voltage, asynchronous circuits are more general than synchronous circuits because they can use any kind of feedback. However, synchronous circuits have proved to be easier to design and use than asynchronous circuits, just as digital are easier than analog circuits. Despite decades of research on asynchronous circuits, virtually all digital systems are essentially synchronous.

Of course, asynchronous circuits are occasionally necessary when communicating between systems with different clocks or when receiving inputs at arbitrary times, just as analog circuits are necessary when communicating with the real world of continuous voltages. Furthermore, research in asynchronous circuits continues to generate interesting insights, some of which can improve synchronous circuits too.

3.4 FINITE STATE MACHINES

Synchronous sequential circuits can be drawn in the forms shown in Figure 3.22. These forms are called *finite state machines* (*FSMs*). They get their name because a circuit with k registers can be in one of a finite number (2^k) of unique states. An FSM has M inputs, N outputs, and k bits of state. It also receives a clock and, optionally, a reset signal. An FSM consists of two blocks of combinational logic, *next state logic* and *output logic*, and a register that stores the state. On each clock edge, the FSM advances to the next state, which was computed based on the current state and inputs. There are two general classes of finite state machines, characterized by their functional specifications. In *Moore machines*, the outputs depend only on the current state of the machine. In *Mealy machines*, the outputs depend on both the current state and the current inputs. Finite state machines provide a systematic way to design synchronous sequential circuits given a functional specification. This method will be explained in the remainder of this section, starting with an example.

3.4.1 FSM Design Example

To illustrate the design of FSMs, consider the problem of inventing a controller for a traffic light at a busy intersection on campus. Engineering students are moseying between their dorms and the labs on Academic Ave. They are busy reading about FSMs in their favorite textbook and aren't

Moore and Mealy machines are named after their promoters, researchers who developed *automata theory*, the mathematical underpinnings of state machines, at Bell Labs.

Edward F. Moore (1925–2003), not to be confused with Intel founder Gordon Moore, published his seminal article, *Gedanken-experiments on Sequential Machines* in 1956. He subsequently became a professor of mathematics and computer science at the University of Wisconsin.

George H. Mealy published *A Method of Synthesizing Sequential Circuits* in 1955. He subsequently wrote the first Bell Labs operating system for the IBM 704 computer. He later joined Harvard University.

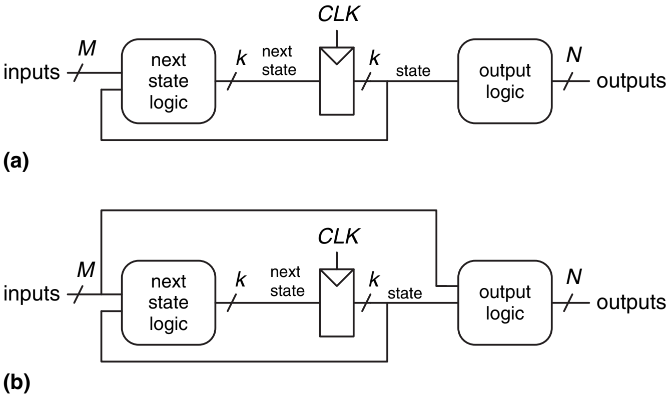


Figure 3.22 Finite state machines: (a) Moore machine, (b) Mealy machine

looking where they are going. Football players are hustling between the athletic fields and the dining hall on Bravado Boulevard. They are tossing the ball back and forth and aren't looking where they are going either. Several serious injuries have already occurred at the intersection of these two roads, and the Dean of Students asks Ben Bitdiddle to install a traffic light before there are fatalities.

Ben decides to solve the problem with an FSM. He installs two traffic sensors, T_A and T_B, on Academic Ave. and Bravado Blvd., respectively. Each sensor indicates TRUE if students are present and FALSE if the street is empty. He also installs two traffic lights, L_A and L_B, to control traffic. Each light receives digital inputs specifying whether it should be green, yellow, or red. Hence, his FSM has two inputs, T_A and T_B, and two outputs, L_A and L_B. The intersection with lights and sensors is shown in Figure 3.23. Ben provides a clock with a 5-second period. On each clock tick (rising edge), the lights may change based on the traffic sensors. He also provides a reset button so that Physical Plant technicians can put the controller in a known initial state when they turn it on. Figure 3.24 shows a black box view of the state machine.

Ben's next step is to sketch the *state transition diagram*, shown in Figure 3.25, to indicate all the possible states of the system and the transitions between these states. When the system is reset, the lights are green on Academic Ave. and red on Bravado Blvd. Every 5 seconds, the controller examines the traffic pattern and decides what to do next. As long as

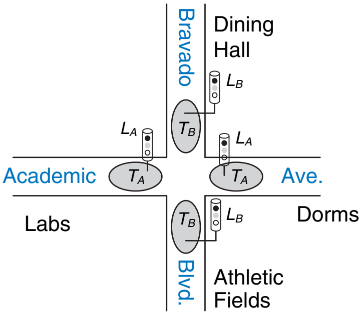


Figure 3.23 **Campus map**

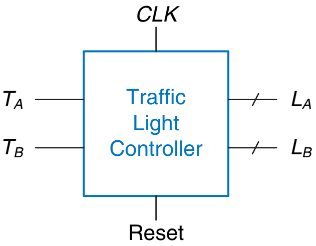


Figure 3.24 **Black box view of finite state machine**

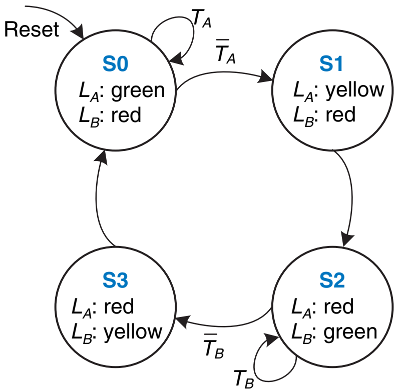


Figure 3.25 **State transition diagram**

traffic is present on Academic Ave., the lights do not change. When there is no longer traffic on Academic Ave., the light on Academic Ave. becomes yellow for 5 seconds before it turns red and Bravado Blvd.'s light turns green. Similarly, the Bravado Blvd. light remains green as long as traffic is present on the boulevard, then turns yellow and eventually red.

In a state transition diagram, circles represent states and arcs represent transitions between states. The transitions take place on the rising edge of the clock; we do not bother to show the clock on the diagram, because it is always present in a synchronous sequential circuit. Moreover, the clock simply controls when the transitions should occur, whereas the diagram indicates which transitions occur. The arc labeled Reset pointing from outer space into state S0 indicates that the system should enter that state upon reset, regardless of what previous state it was in. If a state has multiple arcs leaving it, the arcs are labeled to show what input triggers each transition. For example, when in state S0, the system will remain in that state if T_A is TRUE and move to S1 if T_A is FALSE. If a state has a single arc leaving it, that transition always occurs regardless of the inputs. For example, when in state S1, the system will always move to S2. The value that the outputs have while in a particular state are indicated in the state. For example, while in state S2, L_A is red and L_B is green.

Ben rewrites the state transition diagram as a *state transition table* (Table 3.1), which indicates, for each state and input, what the next state, S', should be. Note that the table uses don't care symbols (X) whenever the next state does not depend on a particular input. Also note that Reset is omitted from the table. Instead, we use resettable flip-flops that always go to state S0 on reset, independent of the inputs.

The state transition diagram is abstract in that it uses states labeled {S0, S1, S2, S3} and outputs labeled {red, yellow, green}. To build a real circuit, the states and outputs must be assigned *binary encodings*. Ben chooses the simple encodings given in Tables 3.2 and 3.3. Each state and each output is encoded with two bits: $S_{1:0}$, $L_{A1:0}$, and $L_{B1:0}$.

Notice that states are designated as S0, S1, etc. The subscripted versions, S_0, S_1, etc., refer to the state bits.

Table 3.1 State transition table

Current State S	Inputs T_A	T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Table 3.2 State encoding

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Table 3.3 Output encoding

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

Ben updates the state transition table to use these binary encodings, as shown in Table 3.4. The revised state transition table is a truth table specifying the next state logic. It defines next state, S', as a function of the current state, S, and the inputs.

From this table, it is straightforward to read off the Boolean equations for the next state in sum-of-products form.

$$\begin{aligned} S'_1 &= \overline{S}_1 S_0 + S_1 \overline{S}_0 \overline{T}_B + S_1 \overline{S}_0 T_B \\ S'_0 &= \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B \end{aligned} \tag{3.1}$$

The equations can be simplified using Karnaugh maps, but often doing it by inspection is easier. For example, the T_B and $\overline{T}_B$ terms in the S'_1 equation are clearly redundant. Thus S'_1 reduces to an XOR operation. Equation 3.2 gives the simplified *next state equations*.

Table 3.4 State transition table with binary encodings

Current State S_1	S_0	Inputs T_A	T_B	Next State S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Table 3.5 **Output table**

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$\begin{aligned} S'_1 &= S_1 \oplus S_0 \\ S'_0 &= \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B \end{aligned} \tag{3.2}$$

Similarly, Ben writes an *output table* (Table 3.5) indicating, for each state, what the output should be in that state. Again, it is straightforward to read off and simplify the Boolean equations for the outputs. For example, observe that L_{A1} is TRUE only on the rows where S_1 is TRUE.

$$\begin{aligned} L_{A1} &= S_1 \\ L_{A0} &= \overline{S}_1 S_0 \\ L_{B1} &= \overline{S}_1 \\ L_{B0} &= S_1 S_0 \end{aligned} \tag{3.3}$$

Finally, Ben sketches his Moore FSM in the form of Figure 3.22(a). First, he draws the 2-bit state register, as shown in Figure 3.26(a). On each clock edge, the state register copies the next state, $S'_{1:0}$, to become the state $S_{1:0}$. The state register receives a synchronous or asynchronous reset to initialize the FSM at startup. Then, he draws the next state logic, based on Equation 3.2, which computes the next state from the current state and inputs, as shown in Figure 3.26(b). Finally, he draws the output logic, based on Equation 3.3, which computes the outputs from the current state, as shown in Figure 3.26(c).

Figure 3.27 shows a timing diagram illustrating the traffic light controller going through a sequence of states. The diagram shows *CLK*, Reset, the inputs T_A and T_B, next state S', state S, and outputs L_A and L_B. Arrows indicate causality; for example, changing the state causes the outputs to change, and changing the inputs causes the next state to change. Dashed lines indicate the rising edges of *CLK* when the state changes.

The clock has a 5-second period, so the traffic lights change at most once every 5 seconds. When the finite state machine is first turned on, its state is unknown, as indicated by the question marks. Therefore, the system should be reset to put it into a known state. In this timing diagram, S

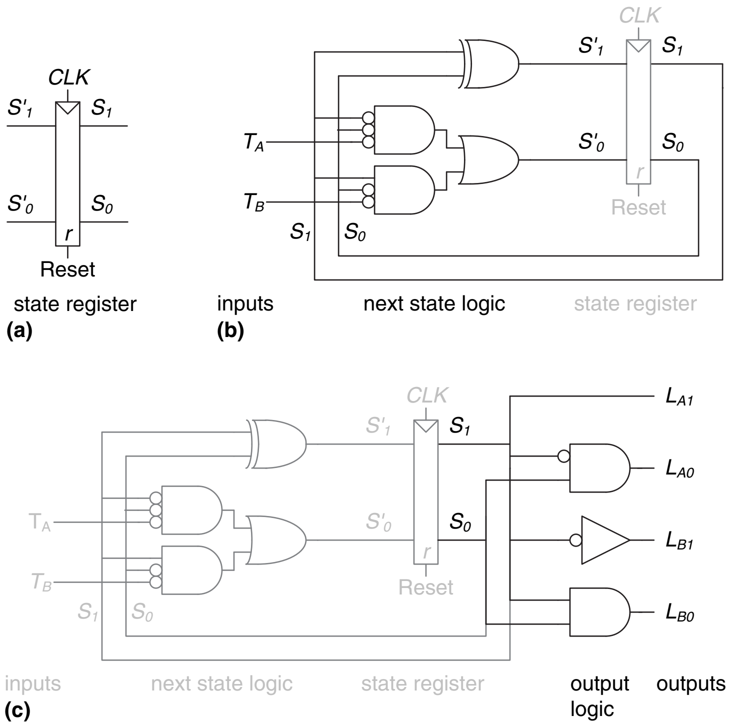

CLK
S'_1
S_1
S'_0
S_0
r
Reset
state register
(a)
T_A
T_B
S_1
S_0
CLK
S'_1
S_1
S'_0
S_0
r
Reset
inputs
next state logic
state register
(b)
T_A
T_B
S_1
S_0
CLK
S'_1
S_1
S'_0
S_0
r
Reset
L_{A1}
L_{A0}
L_{B1}
L_{B0}
inputs
next state logic
state register
output logic
outputs
(c)

Figure 3.26 State machine circuit for traffic light controller

This schematic uses some AND gates with bubbles on the inputs. They might be constructed with AND gates and input inverters, with NOR gates and inverters for the non-bubbled inputs, or with some other combination of gates. The best choice depends on the particular implementation technology.

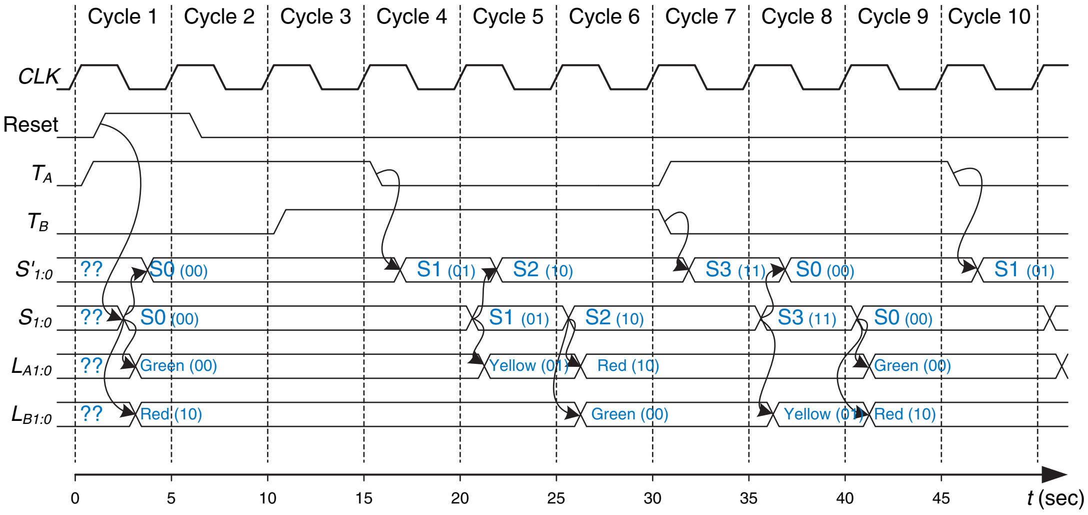
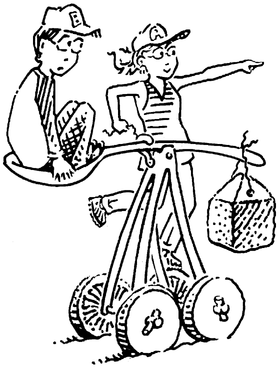

Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Cycle 8
Cycle 9
Cycle 10
CLK
Reset
T_A
T_B
$S'_{1:0}$
$S_{1:0}$
$L_{A1:0}$
$L_{B1:0}$
??
S0 (00)
S1 (01)
S2 (10)
S3 (11)
S0 (00)
S1 (01)
??
S0 (00)
S1 (01)
S2 (10)
S3 (11)
S0 (00)
??
Green (00)
Yellow (01)
Red (10)
Green (00)
??
Red (10)
Green (00)
Yellow (01)
Red (10)
0
5
10
15
20
25
30
35
40
45
t (sec)

Figure 3.27 Timing diagram for traffic light controller

immediately resets to S0, indicating that asynchronously resettable flip-flops are being used. In state S0, light L_A is green and light L_B is red.

In this example, traffic arrives immediately on Academic Ave. Therefore, the controller remains in state S0, keeping L_A green even though traffic arrives on Bravado Blvd. and starts waiting. After 15 seconds, the traffic on Academic Ave. has all passed through and T_A falls. At the following clock edge, the controller moves to state S1, turning L_A yellow. In another 5 seconds, the controller proceeds to state S2 in which L_A turns red and L_B turns green. The controller waits in state S2 until all the traffic on Bravado Blvd. has passed through. It then proceeds to state S3, turning L_B yellow. 5 seconds later, the controller enters state S0, turning L_B red and L_A green. The process repeats.

Despite Ben's best efforts, students don't pay attention to traffic lights and collisions continue to occur. The Dean of Students next asks him and Alyssa to design a catapult to throw engineering students directly from their dorm roofs through the open windows of the lab, bypassing the troublesome intersection all together. But that is the subject of another textbook.



3.4.2 State Encodings

In the previous example, the state and output encodings were selected arbitrarily. A different choice would have resulted in a different circuit. A natural question is how to determine the encoding that produces the circuit with the fewest logic gates or the shortest propagation delay. Unfortunately, there is no simple way to find the best encoding except to try all possibilities, which is infeasible when the number of states is large. However, it is often possible to choose a good encoding by inspection, so that related states or outputs share bits. Computer-aided design (CAD) tools are also good at searching the set of possible encodings and selecting a reasonable one.

One important decision in state encoding is the choice between binary encoding and one-hot encoding. With *binary encoding*, as was used in the traffic light controller example, each state is represented as a binary number. Because K binary numbers can be represented by $\log_2 K$ bits, a system with K states only needs $\log_2 K$ bits of state.

In *one-hot encoding*, a separate bit of state is used for each state. It is called one-hot because only one bit is "hot" or TRUE at any time. For example, a one-hot encoded FSM with three states would have state encodings of 001, 010, and 100. Each bit of state is stored in a flip-flop, so one-hot encoding requires more flip-flops than binary encoding. However, with one-hot encoding, the next-state and output logic is often simpler, so fewer gates are required. The best encoding choice depends on the specific FSM.

Example 3.6 FSM STATE ENCODING

A *divide-by-N counter* has one output and no inputs. The output Y is HIGH for one clock cycle out of every N. In other words, the output divides the frequency of the clock by N. The waveform and state transition diagram for a divide-by-3 counter is shown in Figure 3.28. Sketch circuit designs for such a counter using binary and one-hot state encodings.

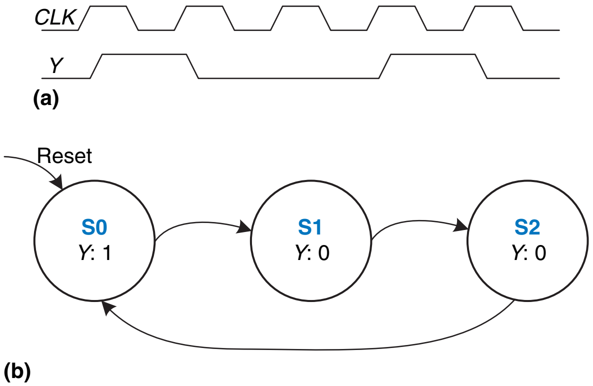


Figure 3.28 **Divide-by-3 counter (a) waveform and (b) state transition diagram**

Table 3.6 **Divide-by-3 counter state transition table**

Current State	Next State
S0	S1
S1	S2
S2	S0

Table 3.7 **Divide-by-3 counter output table**

Current State	Output
S0	1
S1	0
S2	0

Solution: Tables 3.6 and 3.7 show the abstract state transition and output tables before encoding.

Table 3.8 compares binary and one-hot encodings for the three states.

The binary encoding uses two bits of state. Using this encoding, the state transition table is shown in Table 3.9. Note that there are no inputs; the next state depends only on the current state. The output table is left as an exercise to the reader. The next-state and output equations are:

$$\begin{aligned} S'_1 &= \overline{S}_1 S_0 \\ S'_0 &= \overline{S}_1 \overline{S}_0 \end{aligned} \tag{3.4}$$

$$Y = \overline{S}_1 \overline{S}_0 \tag{3.5}$$

The one-hot encoding uses three bits of state. The state transition table for this encoding is shown in Table 3.10 and the output table is again left as an exercise to the reader. The next-state and output equations are as follows:

$$\begin{aligned} S'_2 &= S_1 \\ S'_1 &= S_0 \\ S'_0 &= S_2 \end{aligned} \tag{3.6}$$

$$Y = S_0 \tag{3.7}$$

Figure 3.29 shows schematics for each of these designs. Note that the hardware for the binary encoded design could be optimized to share the same gate for Y and S'_0. Also observe that the one-hot encoding requires both settable (s) and resettable (r) flip-flops to initialize the machine to S0 on reset. The best implementation choice depends on the relative cost of gates and flip-flops, but the one-hot design is usually preferable for this specific example.

A related encoding is the *one-cold* encoding, in which K states are represented with K bits, exactly one of which is FALSE.

Table 3.8 One-hot and binary encodings for divide-by-3 counter

State	One-Hot Encoding			Binary Encoding	
	S_2	S_1	S_0	S_1	S_0
S0	0	0	1	0	0
S1	0	1	0	0	1
S2	1	0	0	1	0

Table 3.9 State transition table with binary encoding

Current State		Next State	
S_1	S_0	S'_1	S'_0
0	0	0	1
0	1	1	0
1	0	0	0

Table 3.10 State transition table with one-hot encoding

Current State			Next State		
S_2	S_1	S_0	S'_2	S'_1	S'_0
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1

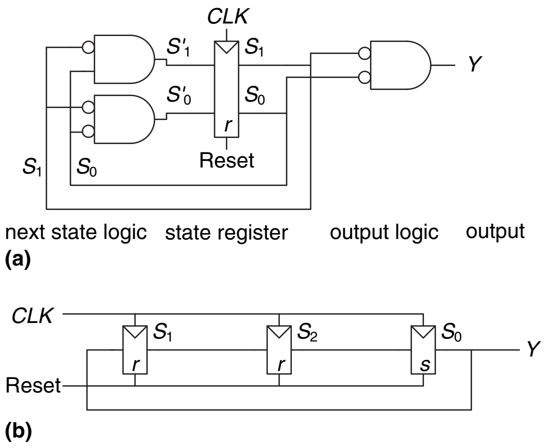

CLK
S'_1
S_1
S'_0
S_0
r
Reset
Y
S_1
S_0
next state logic
state register
output logic
output
(a)
CLK
S_1
S_2
S_0
r
r
s
Y
Reset
(b)

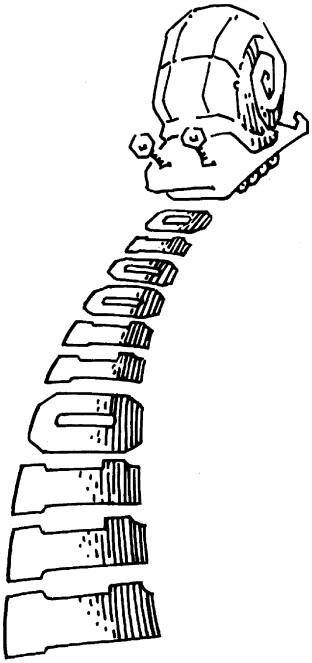
Figure 3.29 Divide-by-3 circuits for (a) binary and (b) one-hot encodings

3.4.3 Moore and Mealy Machines

An easy way to remember the difference between the two types of finite state machines is that a Moore machine typically has *more* states than a Mealy machine for a given problem.

So far, we have shown examples of Moore machines, in which the output depends only on the state of the system. Hence, in state transition diagrams for Moore machines, the outputs are labeled in the circles. Recall that Mealy machines are much like Moore machines, but the outputs can depend on inputs as well as the current state. Hence, in state transition diagrams for Mealy machines, the outputs are labeled on the arcs instead of in the circles. The block of combinational logic that computes the outputs uses the current state and inputs, as was shown in Figure 3.22(b).

Example 3.7 MOORE VERSUS MEALY MACHINES



Alyssa P. Hacker owns a pet robotic snail with an FSM brain. The snail crawls from left to right along a paper tape containing a sequence of 1's and 0's. On each clock cycle, the snail crawls to the next bit. The snail smiles when the last two bits that it has crawled over are, from left to right, 01. Design the FSM to compute when the snail should smile. The input *A* is the bit underneath the snail's antennae. The output *Y* is TRUE when the snail smiles. Compare Moore and Mealy state machine designs. Sketch a timing diagram for each machine showing the input, states, and output as Alyssa's snail crawls along the sequence 0100110111.

Solution: The Moore machine requires three states, as shown in Figure 3.30(a). Convince yourself that the state transition diagram is correct. In particular, why is there an arc from S2 to S1 when the input is 0?

In comparison, the Mealy machine requires only two states, as shown in Figure 3.30(b). Each arc is labeled as *A*/*Y*. *A* is the value of the input that causes that transition, and *Y* is the corresponding output.

Tables 3.11 and 3.12 show the state transition and output tables for the Moore machine. The Moore machine requires at least two bits of state. Consider using a binary state encoding: S0 = 00, S1 = 01, and S2 = 10. Tables 3.13 and 3.14 rewrite the state transition and output tables with these encodings.

From these tables, we find the next state and output equations by inspection. Note that these equations are simplified using the fact that state 11 does not exist. Thus, the corresponding next state and output for the non-existent state are don't cares (not shown in the tables). We use the don't cares to minimize our equations.

$$\begin{aligned} S'_1 &= S_0 A \\ S'_0 &= \overline{A} \end{aligned} \tag{3.8}$$

$$Y = S_1 \tag{3.9}$$

Table 3.15 shows the combined state transition and output table for the Mealy machine. The Mealy machine requires only one bit of state. Consider using a binary state encoding: S0 = 0 and S1 = 1. Table 3.16 rewrites the state transition and output table with these encodings.

From these tables, we find the next state and output equations by inspection.

$$S'_0 = \overline{A} \tag{3.10}$$

$$Y = S_0 A \tag{3.11}$$

The Moore and Mealy machine schematics are shown in Figure 3.31. The timing diagrams for each machine are shown in Figure 3.32 (see page 135). The two machines follow a different sequence of states. Moreover, the Mealy machine's output rises a cycle sooner because it responds to the input rather than waiting for the state change. If the Mealy output were delayed through a flip-flop, it would match the Moore output. When choosing your FSM design style, consider when you want your outputs to respond.

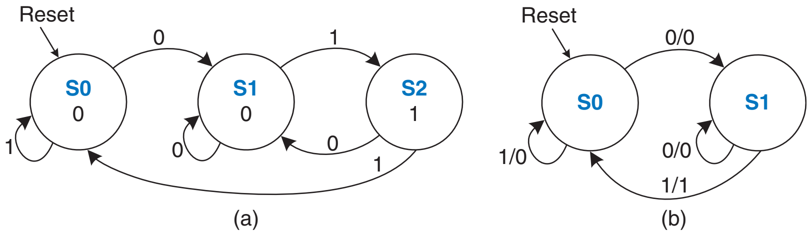


Figure 3.30 FSM state transition diagrams: (a) Moore machine, (b) Mealy machine

Table 3.11 Moore state transition table

Current State S	Input A	Next State S'
S0	0	S1
S0	1	S0
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S0

Table 3.12 Moore output table

Current State S	Output Y
S0	0
S1	0
S2	1

Table 3.13 Moore state transition table with state encodings

Current State S_1	Current State S_0	Input A	Next State S'_1	Next State S'_0
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

Table 3.14 Moore output table with state encodings

Current State S_1	Current State S_0	Output Y
0	0	0
0	1	0
1	0	1

Table 3.15 Mealy state transition and output table

Current State S	Input A	Next State S'	Output Y
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

Table 3.16 Mealy state transition and output table with state encodings

Current State S_0	Input A	Next State S'_0	Output Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

3.4.4 Factoring State Machines

Designing complex FSMs is often easier if they can be broken down into multiple interacting simpler state machines such that the output of some machines is the input of others. This application of hierarchy and modularity is called *factoring* of state machines.

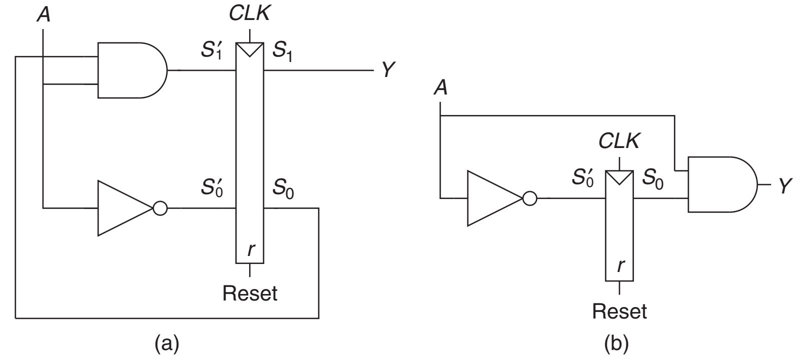


Figure 3.31 **FSM schematics for (a) Moore and (b) Mealy machines**

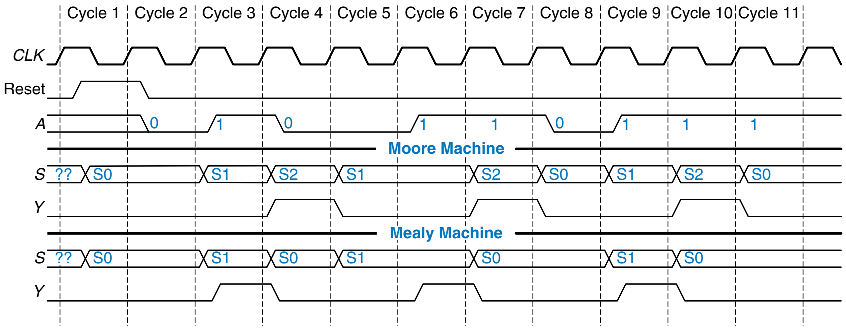


Figure 3.32 **Timing diagrams for Moore and Mealy machines**

Example 3.8 UNFACTORED AND FACTORED STATE MACHINES

Modify the traffic light controller from Section 3.4.1 to have a parade mode, which keeps the Bravado Boulevard light green while spectators and the band march to football games in scattered groups. The controller receives two more inputs: P and R. Asserting P for at least one cycle enters parade mode. Asserting R for at least one cycle leaves parade mode. When in parade mode, the controller proceeds through its usual sequence until L_B turns green, then remains in that state with L_B green until parade mode ends.

First, sketch a state transition diagram for a single FSM, as shown in Figure 3.33(a). Then, sketch the state transition diagrams for two interacting FSMs, as shown in Figure 3.33(b). The Mode FSM asserts the output M when it is in parade mode. The Lights FSM controls the lights based on M and the traffic sensors, T_A and T_B.

Solution: Figure 3.34(a) shows the single FSM design. States S0 to S3 handle normal mode. States S4 to S7 handle parade mode. The two halves of the diagram are almost identical, but in parade mode, the FSM remains in S6 with a green light on Bravado Blvd. The P and R inputs control movement between these two halves. The FSM is messy and tedious to design. Figure 3.34(b) shows the factored FSM design. The mode FSM has two states to track whether the lights are in normal or parade mode. The Lights FSM is modified to remain in S2 while M is TRUE.

Figure 3.33 **(a) single and (b) factored designs for modified traffic light controller FSM**

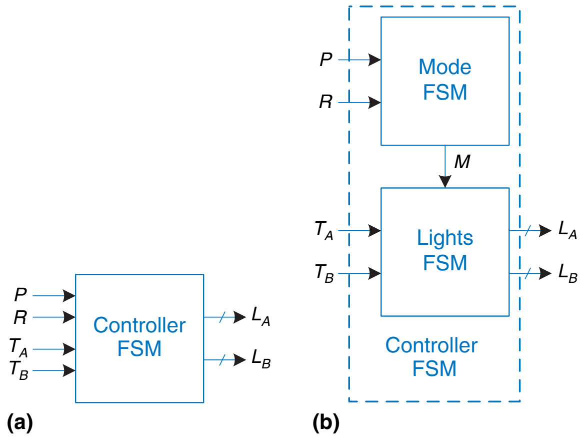
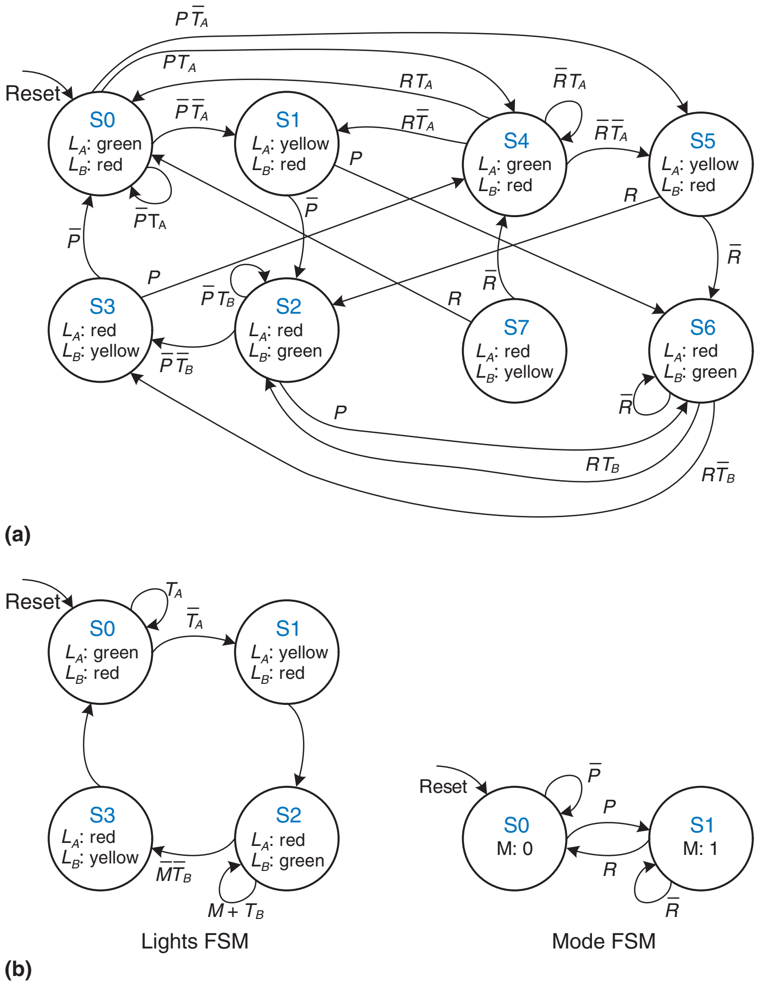

P
R
T_A
T_B
Controller FSM
L_A
L_B
(a)
Mode FSM
M
Lights FSM
Controller FSM
P
R
T_A
T_B
L_A
L_B
(b)

Figure 3.34 **State transition diagrams: (a) unfactored, (b) factored**


Reset
S0
L_A: green
L_B: red
S1
L_A: yellow
L_B: red
S2
L_A: red
L_B: green
S3
L_A: red
L_B: yellow
S4
L_A: green
L_B: red
S5
L_A: yellow
L_B: red
S6
L_A: red
L_B: green
S7
L_A: red
L_B: yellow
$P\bar{T}_A$
PT_A
$\bar{P}\bar{T}_A$
$\bar{P}T_A$
RT_A
$R\bar{T}_A$
$\bar{R}T_A$
$\bar{R}\bar{T}_A$
P
$\bar{P}$
$\bar{P}T_B$
$\bar{P}\bar{T}_B$
R
$\bar{R}$
RT_B
$R\bar{T}_B$
(a)
Reset
S0
L_A: green
L_B: red
S1
L_A: yellow
L_B: red
S2
L_A: red
L_B: green
S3
L_A: red
L_B: yellow
T_A
$\bar{T}_A$
$\bar{M}\bar{T}_B$
$M + T_B$
Lights FSM
Reset
S0
M: 0
S1
M: 1
$\bar{P}$
P
R
$\bar{R}$
Mode FSM
(b)

3.4.5 Deriving an FSM from a Schematic

Deriving the state transition diagram from a schematic follows nearly the reverse process of FSM design. This process can be necessary, for example, when taking on an incompletely documented project or reverse engineering somebody else's system.

- Examine circuit, stating inputs, outputs, and state bits.
- Write next state and output equations.
- Create next state and output tables.
- Reduce the next state table to eliminate unreachable states.
- Assign each valid state bit combination a name.
- Rewrite next state and output tables with state names.
- Draw state transition diagram.
- State in words what the FSM does.

In the final step, be careful to succinctly describe the overall purpose and function of the FSM—do not simply restate each transition of the state transition diagram.

Example 3.9 DERIVING AN FSM FROM ITS CIRCUIT

Alyssa P. Hacker arrives home, but her keypad lock has been rewired and her old code no longer works. A piece of paper is taped to it showing the circuit diagram in Figure 3.35. Alyssa thinks the circuit could be a finite state machine and decides to derive the state transition diagram to see if it helps her get in the door.

Solution: Alyssa begins by examining the circuit. The input is $A_{1:0}$ and the output is *Unlock*. The state bits are already labeled in Figure 3.35. This is a Moore

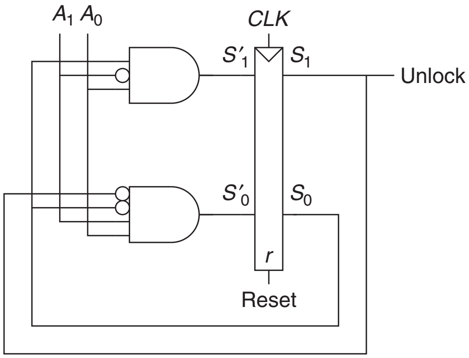


Figure 3.35 Circuit of found FSM for Example 3.9

machine because the output depends only on the state bits. From the circuit, she writes down the next state and output equations directly:

$$\begin{aligned} S'_1 &= S_0\overline{A_1}A_0 \\ S'_0 &= \overline{S_1}\,\overline{S_0}A_1A_0 \\ Unlock &= S_1 \end{aligned} \tag{3.12}$$

Next, she writes down the next state and output tables from the equations, as shown in Tables 3.17 and 3.18, first placing 1's in the tables as indicated by Equation 3.12. She places 0's everywhere else.

Alyssa reduces the table by removing unused states and combining rows using don't cares. The $S_{1:0} = 11$ state is never listed as a possible next state in Table 3.17, so rows with this current state are removed. For current state $S_{1:0} = 10$, the next

Table 3.17 Next state table derived from circuit in Figure 3.35

Current State		Input		Next State	
S_1	S_0	A_1	A_0	S'_1	S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	**1**
0	1	0	0	0	0
0	1	0	1	**1**	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	**1**	0
1	1	1	0	0	0
1	1	1	1	0	0

Table 3.18 Output table derived from circuit in Figure 3.35

Current State		Output
S_1	S_0	$Unlock$
0	0	0
0	1	0
1	0	**1**
1	1	**1**

Table 3.19 Reduced next state table

Current State S_1	S_0	Input A_1	A_0	Next State S'_1	S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	X	X	0	0

Table 3.20 Reduced output table

Current State S_1	S_0	Output *Unlock*
0	0	0
0	1	0
1	0	1

Table 3.21 Symbolic next state table

Current State S	Input A	Next State S'
S0	0	S0
S0	1	S0
S0	2	S0
S0	3	S1
S1	0	S0
S1	1	S2
S1	2	S0
S1	3	S0
S2	X	S0

Table 3.22 Symbolic output table

Current State S	Output *Unlock*
S0	0
S1	0
S2	1

state is always $S_{1:0} = 00$, independent of the inputs, so don't cares are inserted for the inputs. The reduced tables are shown in Tables 3.19 and 3.20.

She assigns names to each state bit combination: S0 is $S_{1:0} = 00$, S1 is $S_{1:0} = 01$, and S2 is $S_{1:0} = 10$. Tables 3.21 and 3.22 show the next state and output tables with state names.

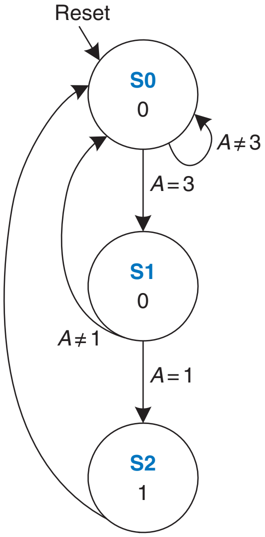


Figure 3.36 **State transition diagram of found FSM from Example 3.9**

Alyssa writes down the state transition diagram shown in Figure 3.36 using Tables 3.21 and 3.22. By inspection, she can see that the finite state machine unlocks the door only after detecting an input value, $A_{1:0}$, of three followed by an input value of one. The door is then locked again. Alyssa tries this code on the door key pad and the door opens!

3.4.6 FSM Review

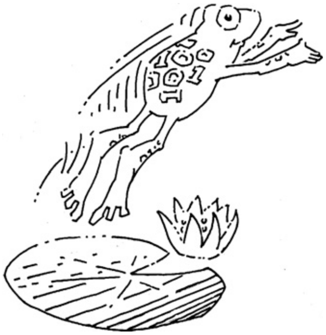
Finite state machines are a powerful way to systematically design sequential circuits from a written specification. Use the following procedure to design an FSM:

- Identify the inputs and outputs.
- Sketch a state transition diagram.
- For a Moore machine:
  - Write a state transition table.
  - Write an output table.
- For a Mealy machine:
  - Write a combined state transition and output table.
- Select state encodings—your selection affects the hardware design.
- Write Boolean equations for the next state and output logic.
- Sketch the circuit schematic.

We will repeatedly use FSMs to design complex digital systems throughout this book.

3.5 TIMING OF SEQUENTIAL LOGIC

Recall that a flip-flop copies the input D to the output Q on the rising edge of the clock. This process is called *sampling* D on the clock edge. If D is *stable* at either 0 or 1 when the clock rises, this behavior is clearly defined. But what happens if D is changing at the same time the clock rises?



This problem is similar to that faced by a camera when snapping a picture. Imagine photographing a frog jumping from a lily pad into the lake. If you take the picture before the jump, you will see a frog on a lily pad. If you take the picture after the jump, you will see ripples in the water. But if you take it just as the frog jumps, you may see a blurred image of the frog stretching from the lily pad into the water. A camera is characterized by its *aperture time*, during which the object must remain still for a sharp image to be captured. Similarly, a sequential element has an aperture time around the clock edge, during which the input must be stable for the flip-flop to produce a well-defined output.

The aperture of a sequential element is defined by a *setup* time and a *hold* time, before and after the clock edge, respectively. Just as the static discipline limited us to using logic levels outside the forbidden zone, the *dynamic discipline* limits us to using signals that change outside the aperture time. By taking advantage of the dynamic discipline, we can think of time in discrete units called clock cycles, just as we think of signal levels as discrete 1's and 0's. A signal may glitch and oscillate wildly for some bounded amount of time. Under the dynamic discipline, we are concerned only about its final value at the end of the clock cycle, after it has settled to a stable value. Hence, we can simply write $A[n]$, the value of signal A at the end of the *nth* clock cycle, where n is an integer, rather than $A(t)$, the value of A at some instant t, where t is any real number.

The clock period has to be long enough for all signals to settle. This sets a limit on the speed of the system. In real systems, the clock does not reach all flip-flops at precisely the same time. This variation in time, called clock skew, further increases the necessary clock period.

Sometimes it is impossible to satisfy the dynamic discipline, especially when interfacing with the real world. For example, consider a circuit with an input coming from a button. A monkey might press the button just as the clock rises. This can result in a phenomenon called metastability, where the flip-flop captures a value partway between 0 and 1 that can take an unlimited amount of time to resolve into a good logic value. The solution to such asynchronous inputs is to use a synchronizer, which has a very small (but nonzero) probability of producing an illegal logic value.

We expand on all of these ideas in the rest of this section.

3.5.1 The Dynamic Discipline

So far, we have focused on the functional specification of sequential circuits. Recall that a synchronous sequential circuit, such as a flip-flop or FSM, also has a timing specification, as illustrated in Figure 3.37. When the clock rises, the output (or outputs) may start to change after the clock-to-Q *contamination delay*, t_{ccq}, and must definitely settle to the final value within the clock-to-Q *propagation delay*, t_{pcq}. These represent the fastest and slowest delays through the circuit, respectively. For the circuit to sample its input correctly, the input (or inputs) must have stabilized at least some *setup time*, t_{setup}, before the rising edge of the clock and must remain stable for at least some *hold time*, t_{hold}, after the rising edge of the clock. The sum of the setup and hold times is called the *aperture time* of the circuit, because it is the total time for which the input must remain stable.

The *dynamic discipline* states that the inputs of a synchronous sequential circuit must be stable during the setup and hold aperture time around the clock edge. By imposing this requirement, we guarantee that the flip-flops sample signals while they are not changing. Because we are concerned only about the final values of the inputs at the time they are sampled, we can treat signals as discrete in time as well as in logic levels.

3.5.2 System Timing

In the three decades from when one of the authors' families bought an Apple II+ computer to the present time of writing, microprocessor clock frequencies have increased from 1 MHz to several GHz, a factor of more than 1000. This speedup partially explains the revolutionary changes computers have made in society.

The *clock period* or *cycle time*, T_c, is the time between rising edges of a repetitive clock signal. Its reciprocal, $f_c = 1/T_c$, is the *clock frequency*. All else being the same, increasing the clock frequency increases the work that a digital system can accomplish per unit time. Frequency is measured in units of Hertz (Hz), or cycles per second: 1 megahertz (MHz) = 10^6 Hz, and 1 gigahertz (GHz) = 10^9 Hz.

Figure 3.38(a) illustrates a generic path in a synchronous sequential circuit whose clock period we wish to calculate. On the rising edge of the clock, register R1 produces output (or outputs) $Q1$. These signals enter a block of combinational logic, producing $D2$, the input (or inputs) to register R2. The timing diagram in Figure 3.38(b) shows that each output signal may start to change a contamination delay after its input

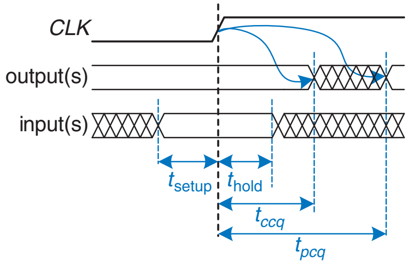


Figure 3.37 **Timing specification for synchronous sequential circuit**

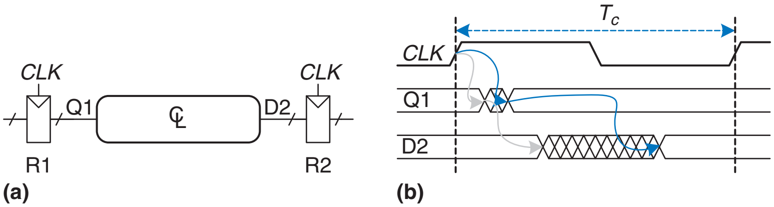


Figure 3.38 **Path between registers and timing diagram**

changes and settles to the final value within a propagation delay after its input settles. The gray arrows represent the contamination delay through R1 and the combinational logic, and the blue arrows represent the propagation delay through R1 and the combinational logic. We analyze the timing constraints with respect to the setup and hold time of the second register, R2.

Setup Time Constraint

Figure 3.39 is the timing diagram showing only the maximum delay through the path, indicated by the blue arrows. To satisfy the setup time of R2, *D*2 must settle no later than the setup time before the next clock edge. Hence, we find an equation for the minimum clock period:

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} \tag{3.13}$$

In commercial designs, the clock period is often dictated by the Director of Engineering or by the marketing department (to ensure a competitive product). Moreover, the flip-flop clock-to-*Q* propagation delay and setup time, t_{pcq} and t_{setup}, are specified by the manufacturer. Hence, we rearrange Equation 3.13 to solve for the maximum propagation delay through the combinational logic, which is usually the only variable under the control of the individual designer.

$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}}) \tag{3.14}$$

The term in parentheses, $t_{pcq} + t_{\text{setup}}$, is called the *sequencing overhead*. Ideally, the entire cycle time T_c would be available for useful

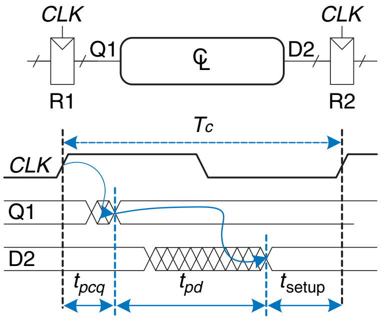


Figure 3.39 **Maximum delay for setup time constraint**

computation in the combinational logic, t_{pd}. However, the sequencing overhead of the flip-flop cuts into this time. Equation 3.14 is called the *setup time constraint* or *max-delay constraint*, because it depends on the setup time and limits the maximum delay through combinational logic.

If the propagation delay through the combinational logic is too great, $D2$ may not have settled to its final value by the time R2 needs it to be stable and samples it. Hence, R2 may sample an incorrect result or even an illegal logic level, a level in the forbidden region. In such a case, the circuit will malfunction. The problem can be solved by increasing the clock period or by redesigning the combinational logic to have a shorter propagation delay.

Hold Time Constraint

The register R2 in Figure 3.38(a) also has a *hold time constraint*. Its input, $D2$, must not change until some time, t_{hold}, after the rising edge of the clock. According to Figure 3.40, $D2$ might change as soon as $t_{ccq} + t_{cd}$ after the rising edge of the clock. Hence, we find

$$t_{ccq} + t_{cd} \geq t_{\text{hold}} \tag{3.15}$$

Again, t_{ccq} and t_{hold} are characteristics of the flip-flop that are usually outside the designer's control. Rearranging, we can solve for the minimum contamination delay through the combinational logic:

$$t_{cd} \geq t_{\text{hold}} - t_{ccq} \tag{3.16}$$

Equation 3.16 is also called the *hold time constraint* or *min-delay constraint* because it limits the minimum delay through combinational logic.

We have assumed that any logic elements can be connected to each other without introducing timing problems. In particular, we would expect that two flip-flops may be directly cascaded as in Figure 3.41 without causing hold time problems.

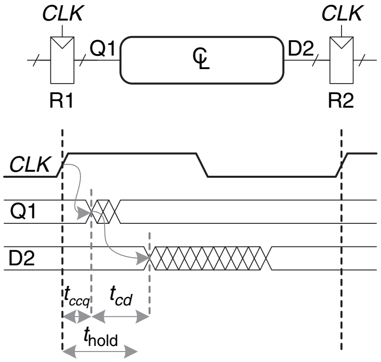


Figure 3.40 Minimum delay for hold time constraint

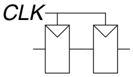


Figure 3.41 **Back-to-back flip-flops**

In such a case, $t_{cd} = 0$ because there is no combinational logic between flip-flops. Substituting into Equation 3.16 yields the requirement that

$$t_{\text{hold}} \leq t_{ccq} \tag{3.17}$$

In other words, a reliable flip-flop must have a hold time shorter than its contamination delay. Often, flip-flops are designed with $t_{\text{hold}} = 0$, so that Equation 3.17 is always satisfied. Unless noted otherwise, we will usually make that assumption and ignore the hold time constraint in this book.

Nevertheless, hold time constraints are critically important. If they are violated, the only solution is to increase the contamination delay through the logic, which requires redesigning the circuit. Unlike setup time constraints, they cannot be fixed by adjusting the clock period. Redesigning an integrated circuit and manufacturing the corrected design takes months and millions of dollars in today's advanced technologies, so *hold time violations* must be taken extremely seriously.

Putting It All Together

Sequential circuits have setup and hold time constraints that dictate the maximum and minimum delays of the combinational logic between flip-flops. Modern flip-flops are usually designed so that the minimum delay through the combinational logic is 0—that is, flip-flops can be placed back-to-back. The maximum delay constraint limits the number of consecutive gates on the critical path of a high-speed circuit, because a high clock frequency means a short clock period.

Example 3.10 TIMING ANALYSIS

Ben Bitdiddle designed the circuit in Figure 3.42. According to the data sheets for the components he is using, flip-flops have a clock-to-Q contamination delay of 30 ps and a propagation delay of 80 ps. They have a setup time of 50 ps and a hold time of 60 ps. Each logic gate has a propagation delay of 40 ps and a

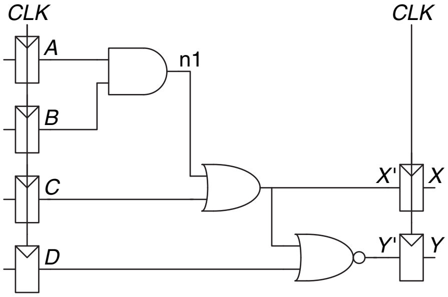


Figure 3.42 **Sample circuit for timing analysis**

contamination delay of 25 ps. Help Ben determine the maximum clock frequency and whether any hold time violations could occur. This process is called *timing analysis*.

Solution: Figure 3.43(a) shows waveforms illustrating when the signals might change. The inputs, A to D, are registered, so they only change shortly after CLK rises.

The critical path occurs when $B = 1$, $C = 0$, $D = 0$, and A rises from 0 to 1, triggering n1 to rise, X' to rise, and Y' to fall, as shown in Figure 3.43(b). This path involves three gate delays. For the critical path, we assume that each gate requires its full propagation delay. Y' must setup before the next rising edge of the CLK. Hence, the minimum cycle time is

$$T_c \geq t_{pcq} + 3\,t_{pd} + t_{\text{setup}} = 80 + 3 \times 40 + 50 = 250\text{ps} \qquad (3.18)$$

The maximum clock frequency is $f_c = 1/T_c = 4$ GHz.

A short path occurs when $A = 0$ and C rises, causing X' to rise, as shown in Figure 3.43(c). For the short path, we assume that each gate switches after only a contamination delay. This path involves only one gate delay, so it may occur after $t_{ccq} + t_{cd} = 30 + 25 = 55$ ps. But recall that the flip-flop has a hold time of

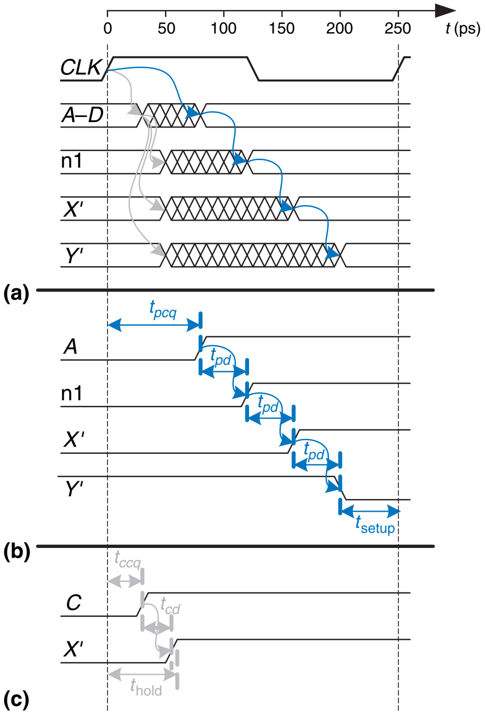


Figure 3.43 Timing diagram: (a) general case, (b) critical path, (c) short path

60 ps, meaning that *X′* must remain stable for 60 ps after the rising edge of *CLK* for the flip-flop to reliably sample its value. In this case, $X' = 0$ at the first rising edge of *CLK*, so we want the flip-flop to capture $X = 0$. Because *X′* did not hold stable long enough, the actual value of *X* is unpredictable. The circuit has a hold time violation and may behave erratically at any clock frequency.

Example 3.11 FIXING HOLD TIME VIOLATIONS

Alyssa P. Hacker proposes to fix Ben's circuit by adding buffers to slow down the short paths, as shown in Figure 3.44. The buffers have the same delays as other gates. Determine the maximum clock frequency and whether any hold time problems could occur.

Solution: Figure 3.45 shows waveforms illustrating when the signals might change. The critical path from *A* to *Y* is unaffected, because it does not pass through any buffers. Therefore, the maximum clock frequency is still 4 GHz. However, the short paths are slowed by the contamination delay of the buffer. Now *X′* will not change until $t_{ccq} + 2t_{cd} = 30 + 2 \times 25 = 80$ ps. This is after the 60 ps hold time has elapsed, so the circuit now operates correctly.

This example had an unusually long hold time to illustrate the point of hold time problems. Most flip-flops are designed with $t_{\text{hold}} < t_{ccq}$ to avoid such problems.

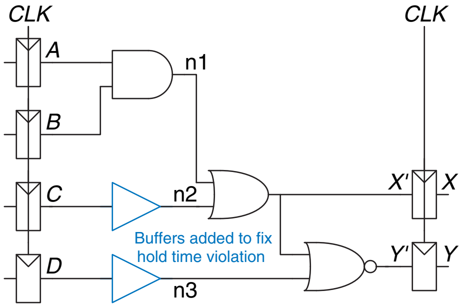


Figure 3.44 Corrected circuit to fix hold time problem

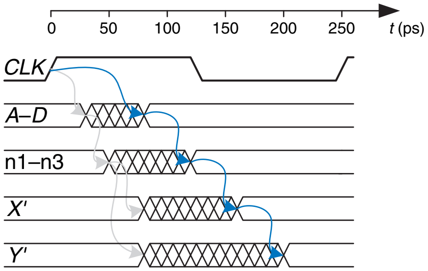


Figure 3.45 Timing diagram with buffers to fix hold time problem

However, some high-performance microprocessors, including the Pentium 4, use an element called a *pulsed latch* in place of a flip-flop. The pulsed latch behaves like a flip-flop but has a short clock-to-Q delay and a long hold time. In general, adding buffers can usually, but not always, solve hold time problems without slowing the critical path.

3.5.3 Clock Skew*

In the previous analysis, we assumed that the clock reaches all registers at exactly the same time. In reality, there is some variation in this time. This variation in clock edges is called *clock skew*. For example, the wires from the clock source to different registers may be of different lengths, resulting in slightly different delays, as shown in Figure 3.46. Noise also results in different delays. Clock gating, described in Section 3.2.5, further delays the clock. If some clocks are gated and others are not, there will be substantial skew between the gated and ungated clocks. In Figure 3.46, *CLK*2 is *early* with respect to *CLK*1, because the clock wire between the two registers follows a scenic route. If the clock had been routed differently, *CLK*1 might have been early instead. When doing timing analysis, we consider the worst-case scenario, so that we can guarantee that the circuit will work under all circumstances.

Figure 3.47 adds skew to the timing diagram from Figure 3.38. The heavy clock line indicates the latest time at which the clock signal might reach any register; the hashed lines show that the clock might arrive up to t_{skew} earlier.

First, consider the setup time constraint shown in Figure 3.48. In the worst case, R1 receives the latest skewed clock and R2 receives the earliest skewed clock, leaving as little time as possible for data to propagate between the registers.

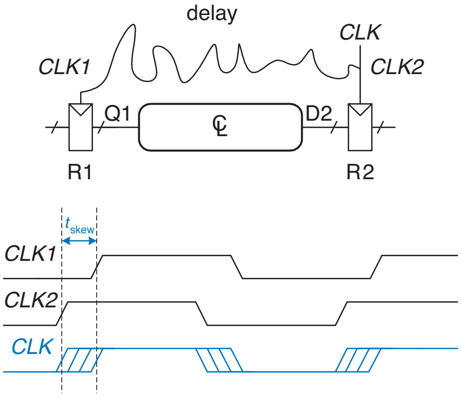


Figure 3.46 Clock skew caused by wire delay

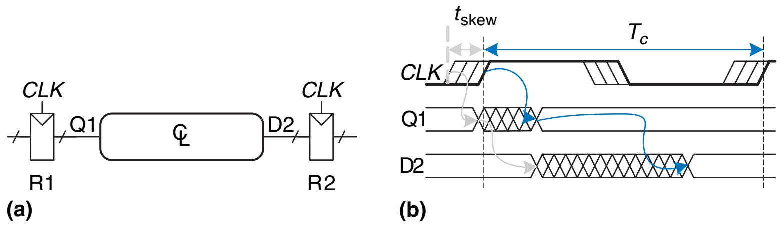


Figure 3.47 **Timing diagram with clock skew**

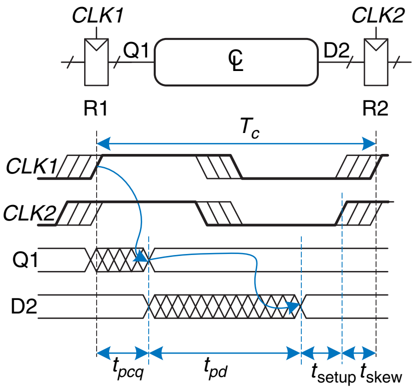


Figure 3.48 **Setup time constraint with clock skew**

The data propagates through the register and combinational logic and must setup before R2 samples it. Hence, we conclude that

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} + t_{\text{skew}} \tag{3.19}$$

$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}} + t_{\text{skew}}) \tag{3.20}$$

Next, consider the hold time constraint shown in Figure 3.49. In the worst case, R1 receives an early skewed clock, *CLK*1, and R2 receives a late skewed clock, *CLK*2. The data zips through the register and combinational logic but must not arrive until a hold time after the late clock. Thus, we find that

$$t_{ccq} + t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} \tag{3.21}$$

$$t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} - t_{ccq} \tag{3.22}$$

In summary, clock skew effectively increases both the setup time and the hold time. It adds to the sequencing overhead, reducing the time available for useful work in the combinational logic. It also increases the required minimum delay through the combinational logic. Even if $t_{\text{hold}} = 0$, a pair of back-to-back flip-flops will violate Equation 3.22 if $t_{\text{skew}} > t_{ccq}$. To prevent

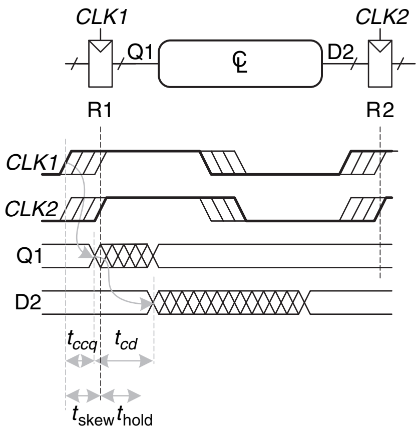


Figure 3.49 **Hold time constraint with clock skew**

serious hold time failures, designers must not permit too much clock skew. Sometimes flip-flops are intentionally designed to be particularly slow (i.e., large t_{ccq}), to prevent hold time problems even when the clock skew is substantial.

Example 3.12 TIMING ANALYSIS WITH CLOCK SKEW

Revisit Example 3.10 and assume that the system has 50 ps of clock skew.

Solution: The critical path remains the same, but the setup time is effectively increased by the skew. Hence, the minimum cycle time is

$$\begin{aligned} T_c &\geq t_{pcq} + 3t_{pd} + t_{\text{setup}} + t_{\text{skew}} \\ &= 80 + 3\times 40 + 50 + 50 = 300\text{ps} \end{aligned} \tag{3.23}$$

The maximum clock frequency is $f_c = 1/T_c = 3.33$ GHz.

The short path also remains the same at 55 ps. The hold time is effectively increased by the skew to $60 + 50 = 110$ ps, which is much greater than 55 ps. Hence, the circuit will violate the hold time and malfunction at any frequency. The circuit violated the hold time constraint even without skew. Skew in the system just makes the violation worse.

Example 3.13 FIXING HOLD TIME VIOLATIONS

Revisit Example 3.11 and assume that the system has 50 ps of clock skew.

Solution: The critical path is unaffected, so the maximum clock frequency remains 3.33 GHz.

The short path increases to 80 ps. This is still less than $t_{\text{hold}} + t_{\text{skew}} = 110$ ps, so the circuit still violates its hold time constraint.

To fix the problem, even more buffers could be inserted. Buffers would need to be added on the critical path as well, reducing the clock frequency. Alternatively, a better flip-flop with a shorter hold time might be used.

3.5.4 Metastability

As noted earlier, it is not always possible to guarantee that the input to a sequential circuit is stable during the aperture time, especially when the input arrives from the external world. Consider a button connected to the input of a flip-flop, as shown in Figure 3.50. When the button is not pressed, $D = 0$. When the button is pressed, $D = 1$. A monkey presses the button at some random time relative to the rising edge of *CLK*. We want to know the output *Q* after the rising edge of *CLK*. In Case I, when the button is pressed much before *CLK*, $Q = 1$. In Case II, when the button is not pressed until long after *CLK*, $Q = 0$. But in Case III, when the button is pressed sometime between t_{setup} before *CLK* and t_{hold} after *CLK*, the input violates the dynamic discipline and the output is undefined.

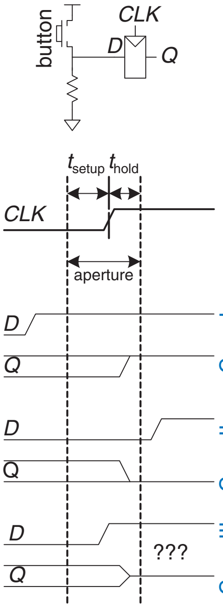


Figure 3.50 Input changing before, after, or during aperture

Metastable State

When a flip-flop samples an input that is changing during its aperture, the output *Q* may momentarily take on a voltage between 0 and V_{DD} that is in the forbidden zone. This is called a *metastable* state. Eventually, the flip-flop will resolve the output to a *stable state* of either 0 or 1. However, the *resolution time* required to reach the stable state is unbounded.

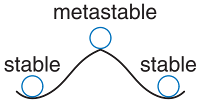


Figure 3.51 Stable and metastable states

The metastable state of a flip-flop is analogous to a ball on the summit of a hill between two valleys, as shown in Figure 3.51. The two valleys are stable states, because a ball in the valley will remain there as long as it is not disturbed. The top of the hill is called metastable because the ball would remain there if it were perfectly balanced. But because nothing is perfect, the ball will eventually roll to one side or the other. The time required for this change to occur depends on how nearly well balanced the ball originally was. Every bistable device has a metastable state between the two stable states.

Resolution Time

If a flip-flop input changes at a random time during the clock cycle, the resolution time, t_{res}, required to resolve to a stable state is also a random variable. If the input changes outside the aperture, then $t_{res} = t_{pcq}$. But if the input happens to change within the aperture, t_{res} can be substantially longer. Theoretical and experimental analyses (see Section 3.5.6) have shown that the probability that the resolution time, t_{res}, exceeds some arbitrary time, t, decreases exponentially with t:

$$P(t_{res} > t) = \frac{T_0}{T_c} e^{-\frac{t}{\tau}} \tag{3.24}$$

where T_c is the clock period, and T_0 and τ are characteristic of the flip-flop. The equation is valid only for t substantially longer than t_{pcq}.

Intuitively, T_0/T_c describes the probability that the input changes at a bad time (i.e., during the aperture time); this probability decreases with the cycle time, T_c. τ is a time constant indicating how fast the flip-flop moves away from the metastable state; it is related to the delay through the cross-coupled gates in the flip-flop.

In summary, if the input to a bistable device such as a flip-flop changes during the aperture time, the output may take on a metastable value for some time before resolving to a stable 0 or 1. The amount of time required to resolve is unbounded, because for any finite time, t, the probability that the flip-flop is still metastable is nonzero. However, this probability drops off exponentially as t increases. Therefore, if we wait long enough, much longer than t_{pcq}, we can expect with exceedingly high probability that the flip-flop will reach a valid logic level.

3.5.5 Synchronizers

Asynchronous inputs to digital systems from the real world are inevitable. Human input is asynchronous, for example. If handled carelessly, these asynchronous inputs can lead to metastable voltages within the system, causing erratic system failures that are extremely difficult to track down and correct. The goal of a digital system designer should be to ensure that, given asynchronous inputs, the probability of encountering a metastable voltage is sufficiently small. "Sufficiently" depends on the context. For a cell phone, perhaps one failure in 10 years is acceptable, because the user can always turn the phone off and back on if it locks up. For a medical device, one failure in the expected life of the universe (10^{10} years) is a better target. To guarantee good logic levels, all asynchronous inputs should be passed through *synchronizers*.

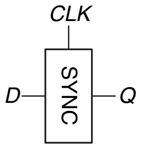


Figure 3.52 **Synchronizer symbol**

A synchronizer, shown in Figure 3.52, is a device that receives an asynchronous input D and a clock CLK. It produces an output Q within a bounded amount of time; the output has a valid logic level with extremely high probability. If D is stable during the aperture, Q should take on the same value as D. If D changes during the aperture, Q may take on either a HIGH or LOW value but must not be metastable.

Figure 3.53 shows a simple way to build a synchronizer out of two flip-flops. F1 samples D on the rising edge of CLK. If D is changing at that time, the output D2 may be momentarily metastable. If the clock

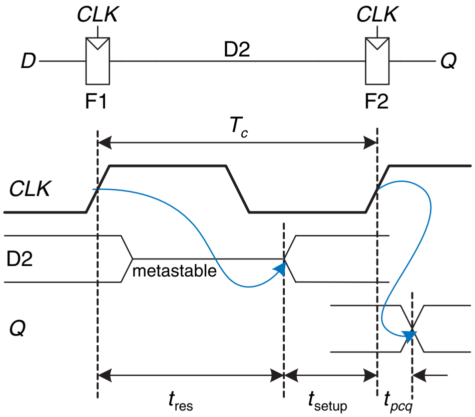


Figure 3.53 **Simple synchronizer**

period is long enough, D2 will, with high probability, resolve to a valid logic level before the end of the period. F2 then samples D2, which is now stable, producing a good output Q.

We say that a synchronizer *fails* if Q, the output of the synchronizer, becomes metastable. This may happen if $D2$ has not resolved to a valid level by the time it must setup at F2—that is, if $t_{res} > T_c - t_{\text{setup}}$. According to Equation 3.24, the probability of failure for a single input change at a random time is

$$P(\text{failure}) = \frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau}} \qquad (3.25)$$

The probability of failure, P(failure), is the probability that the output Q will be metastable upon a single change in D. If D changes once per second, the probability of failure per second is just P(failure). However, if D changes N times per second, the probability of failure per second is N times as great:

$$P(\text{failure})/\text{sec} = N \frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau}} \qquad (3.26)$$

System reliability is usually measured in *mean time between failures* (*MTBF*). As the name suggests, MTBF is the average amount of time between failures of the system. It is the reciprocal of the probability that the system will fail in any given second

$$MTBF = \frac{1}{P(\text{failure})/\text{sec}} = \frac{T_c e^{\frac{T_c - t_{\text{setup}}}{\tau}}}{N T_0} \qquad (3.27)$$

Equation 3.27 shows that the MTBF improves exponentially as the synchronizer waits for a longer time, T_c. For most systems, a synchronizer that waits for one clock cycle provides a safe MTBF. In exceptionally high-speed systems, waiting for more cycles may be necessary.

Example 3.14 SYNCHRONIZER FOR FSM INPUT

The traffic light controller FSM from Section 3.4.1 receives asynchronous inputs from the traffic sensors. Suppose that a synchronizer is used to guarantee stable inputs to the controller. Traffic arrives on average 0.2 times per second. The flip-flops in the synchronizer have the following characteristics: $\tau = 200$ ps, $T_0 = 150$ ps, and $t_{\text{setup}} = 500$ ps. How long must the synchronizer clock period be for the MTBF to exceed 1 year?

Solution: 1 year $\approx \pi \times 10^7$ seconds. Solve Equation 3.27.

$$\pi \times 10^7 = \frac{T_c e^{\frac{T_c - 500\times 10^{-12}}{200\times 10^{-12}}}}{(0.2)(150 \times 10^{-12})} \tag{3.28}$$

This equation has no closed form solution. However, it is easy enough to solve by guess and check. In a spreadsheet, try a few values of T_c and calculate the MTBF until discovering the value of T_c that gives an MTBF of 1 year: $T_c = 3.036$ ns.

3.5.6 Derivation of Resolution Time*

Equation 3.24 can be derived using a basic knowledge of circuit theory, differential equations, and probability. This section can be skipped if you are not interested in the derivation or if you are unfamiliar with the mathematics.

A flip-flop output will be metastable after some time, t, if the flip-flop samples a changing input (causing a metastable condition) and the output does not resolve to a valid level within that time after the clock edge. Symbolically, this can be expressed as

$$P(t_{res} > t) = P(\text{samples changing input}) \times P(\text{unresolved}) \tag{3.29}$$

We consider each probability term individually. The asynchronous input signal switches between 0 and 1 in some time, t_{switch}, as shown in Figure 3.54. The probability that the input changes during the aperture around the clock edge is

$$P(\text{samples changing input}) = \frac{t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}}}{T_c} \tag{3.30}$$

If the flip-flop does enter metastability—that is, with probability P(samples changing input)—the time to resolve from metastability depends on the inner workings of the circuit. This resolution time determines P(unresolved), the probability that the flip-flop has not yet resolved

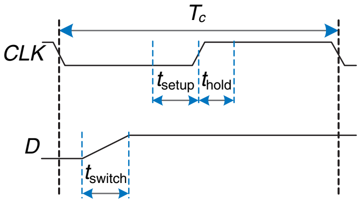


Figure 3.54 Input timing

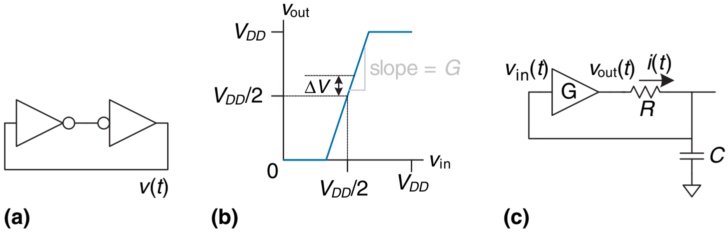


Figure 3.55 Circuit model of bistable device

to a valid logic level after a time t. The remainder of this section analyzes a simple model of a bistable device to estimate this probability.

A bistable device uses storage with positive feedback. Figure 3.55(a) shows this feedback implemented with a pair of inverters; this circuit's behavior is representative of most bistable elements. A pair of inverters behaves like a buffer. Let us model it as having the symmetric DC transfer characteristics shown in Figure 3.55(b), with a slope of G. The buffer can deliver only a finite amount of output current; we can model this as an output resistance, R. All real circuits also have some capacitance C that must be charged up. Charging the capacitor through the resistor causes an RC delay, preventing the buffer from switching instantaneously. Hence, the complete circuit model is shown in Figure 3.55(c), where $v_{out}(t)$ is the voltage of interest conveying the state of the bistable device.

The metastable point for this circuit is $v_{out}(t) = v_{in}(t) = V_{DD}/2$; if the circuit began at exactly that point, it would remain there indefinitely in the absence of noise. Because voltages are continuous variables, the chance that the circuit will begin at exactly the metastable point is vanishingly small. However, the circuit might begin at time 0 near metastability at $v_{out}(0) = V_{DD}/2 + \Delta V$ for some small offset ΔV. In such a case, the positive feedback will eventually drive $v_{out}(t)$ to V_{DD} if $\Delta V > 0$ and to 0 if $\Delta V < 0$. The time required to reach V_{DD} or 0 is the resolution time of the bistable device.

The DC transfer characteristic is nonlinear, but it appears linear near the metastable point, which is the region of interest to us. Specifically, if $v_{in}(t) = V_{DD}/2 + \Delta V/G$, then $v_{out}(t) = V_{DD}/2 + \Delta V$ for small ΔV. The current through the resistor is $i(t) = (v_{out}(t) - v_{in}(t))/R$. The capacitor charges at a rate $dv_{\text{in}}(t)/dt = i(t)/C$. Putting these facts together, we find the governing equation for the output voltage.

$$\frac{dv_{\text{out}}(t)}{dt} = \frac{(G-1)}{RC}\left[v_{\text{out}}(t) - \frac{V_{DD}}{2}\right] \tag{3.31}$$

This is a linear first-order differential equation. Solving it with the initial condition $v_{\text{out}}(0) = V_{DD}/2 + \Delta V$ gives

$$v_{\text{out}}(t) = \frac{V_{DD}}{2} + \Delta V e^{\frac{(G-1)t}{RC}} \tag{3.32}$$

Figure 3.56 plots trajectories for $v_{\text{out}}(t)$ given various starting points. $v_{\text{out}}(t)$ moves exponentially away from the metastable point $V_{DD}/2$ until it saturates at V_{DD} or 0. The output eventually resolves to 1 or 0. The amount of time this takes depends on the initial voltage offset (ΔV) from the metastable point ($V_{DD}/2$).

Solving Equation 3.32 for the resolution time t_{res}, such that $v_{\text{out}}(t_{res}) = V_{DD}$ or 0, gives

$$|\Delta V| e^{\frac{(G-1)t_{res}}{RC}} = \frac{V_{DD}}{2} \tag{3.33}$$

$$t_{res} = \frac{RC}{G-1} \ln \frac{V_{DD}}{2|\Delta V|} \tag{3.34}$$

In summary, the resolution time increases if the bistable device has high resistance or capacitance that causes the output to change slowly. It decreases if the bistable device has high *gain*, G. The resolution time also increases logarithmically as the circuit starts closer to the metastable point ($\Delta V \rightarrow 0$).

Define τ as $\frac{RC}{G-1}$. Solving Equation 3.34 for ΔV finds the initial offset, ΔV_{res}, that gives a particular resolution time, t_{res}:

$$\Delta V_{res} = \frac{V_{DD}}{2} e^{-t_{res}/\tau} \tag{3.35}$$

Suppose that the bistable device samples the input while it is changing. It measures a voltage, $v_{\text{in}}(0)$, which we will assume is uniformly distributed

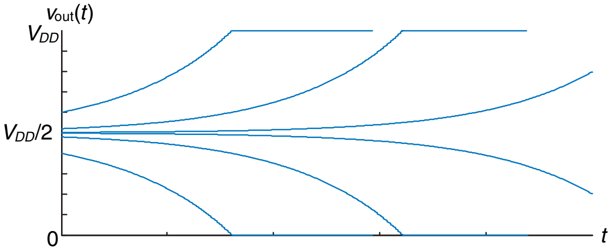


Figure 3.56 Resolution trajectories

between 0 and V_{DD}. The probability that the output has not resolved to a legal value after time t_{res} depends on the probability that the initial offset is sufficiently small. Specifically, the initial offset on v_{out} must be less than ΔV_{res}, so the initial offset on v_{in} must be less than $\Delta V_{res}/G$. Then the probability that the bistable device samples the input at a time to obtain a sufficiently small initial offset is

$$P(\text{unresolved}) = P\left(\left|v_{\text{in}}(0) - \frac{V_{DD}}{2}\right| < \frac{\Delta V_{res}}{G}\right) = \frac{2\Delta V_{res}}{GV_{DD}} \tag{3.36}$$

Putting this all together, the probability that the resolution time exceeds some time t is given by the following equation:

$$P(t_{res} > t) = \frac{t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}}}{GT_c} e^{-\frac{t}{\tau}} \tag{3.37}$$

Observe that Equation 3.37 is in the form of Equation 3.24, where $T_0 = (t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}})/G$ and $\tau = RC/(G-1)$. In summary, we have derived Equation 3.24 and shown how T_0 and τ depend on physical properties of the bistable device.

3.6 PARALLELISM

The speed of a system is characterized by the latency and throughput of information moving through it. We define a *token* to be a group of inputs that are processed to produce a group of outputs. The term conjures up the notion of placing subway tokens on a circuit diagram and moving them around to visualize data moving through the circuit. The *latency* of a system is the time required for one token to pass through the system from start to end. The *throughput* is the number of tokens that can be produced per unit time.

Example 3.15 COOKIE THROUGHPUT AND LATENCY

Ben Bitdiddle is throwing a milk and cookies party to celebrate the installation of his traffic light controller. It takes him 5 minutes to roll cookies and place them on his tray. It then takes 15 minutes for the cookies to bake in the oven. Once the cookies are baked, he starts another tray. What is Ben's throughput and latency for a tray of cookies?

Solution: In this example, a tray of cookies is a token. The latency is 1/3 hour per tray. The throughput is 3 trays/hour.

As you might imagine, the throughput can be improved by processing several tokens at the same time. This is called *parallelism*, and it comes in two forms: spatial and temporal. With *spatial parallelism*, multiple copies of the hardware are provided so that multiple tasks can be done at the same time. With *temporal parallelism*, a task is broken into stages, like an assembly line. Multiple tasks can be spread across the stages. Although each task must pass through all stages, a different task will be in each stage at any given time so multiple tasks can overlap. Temporal parallelism is commonly called *pipelining*. Spatial parallelism is sometimes just called parallelism, but we will avoid that naming convention because it is ambiguous.

Example 3.16 COOKIE PARALLELISM

Ben Bitdiddle has hundreds of friends coming to his party and needs to bake cookies faster. He is considering using spatial and/or temporal parallelism.

Spatial Parallelism: Ben asks Alyssa P. Hacker to help out. She has her own cookie tray and oven.

Temporal Parallelism: Ben gets a second cookie tray. Once he puts one cookie tray in the oven, he starts rolling cookies on the other tray rather than waiting for the first tray to bake.

What is the throughput and latency using spatial parallelism? Using temporal parallelism? Using both?

Solution: The latency is the time required to complete one task from start to finish. In all cases, the latency is 1/3 hour. If Ben starts with no cookies, the latency is the time needed for him to produce the first cookie tray.

The throughput is the number of cookie trays per hour. With spatial parallelism, Ben and Alyssa each complete one tray every 20 minutes. Hence, the throughput doubles, to 6 trays/hour. With temporal parallelism, Ben puts a new tray in the oven every 15 minutes, for a throughput of 4 trays/hour. These are illustrated in Figure 3.57.

If Ben and Alyssa use both techniques, they can bake 8 trays/hour.

Consider a task with latency L. In a system with no parallelism, the throughput is $1/L$. In a spatially parallel system with N copies of the hardware, the throughput is N/L. In a temporally parallel system, the task is ideally broken into N steps, or stages, of equal length. In such a case, the throughput is also N/L, and only one copy of the hardware is required. However, as the cookie example showed, finding N steps of equal length is often impractical. If the longest step has a latency L_1, the pipelined throughput is $1/L_1$.

Pipelining (temporal parallelism) is particularly attractive because it speeds up a circuit without duplicating the hardware. Instead, registers are placed between blocks of combinational logic to divide the logic into shorter stages that can run with a faster clock. The registers prevent a

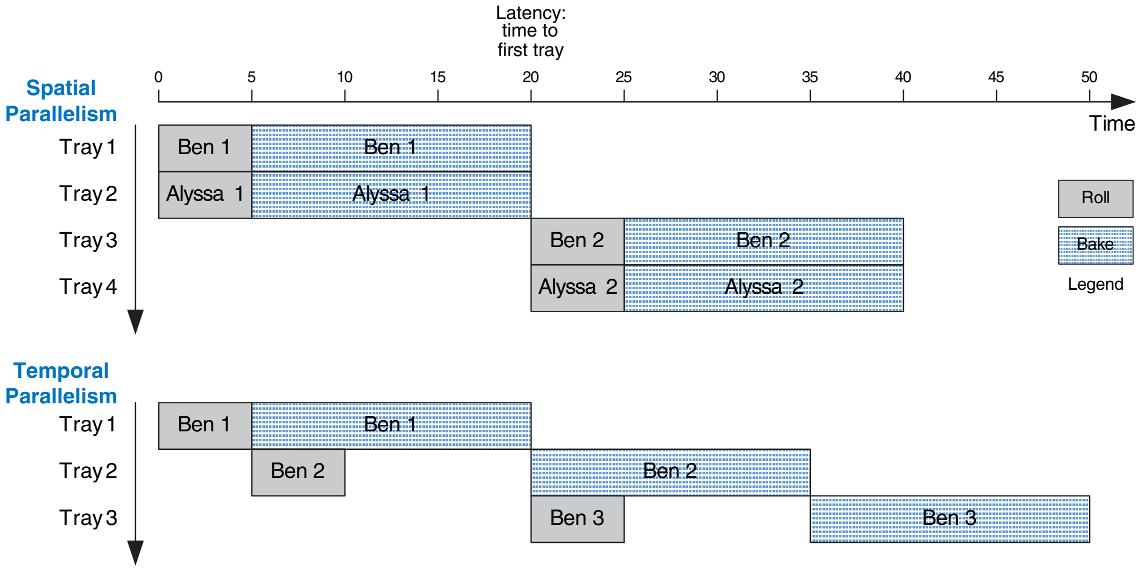


Figure 3.57 **Spatial and temporal parallelism in the cookie kitchen**

token in one pipeline stage from catching up with and corrupting the token in the next stage.

Figure 3.58 shows an example of a circuit with no pipelining. It contains four blocks of logic between the registers. The critical path passes through blocks 2, 3, and 4. Assume that the register has a clock-to-*Q* propagation delay of 0.3 ns and a setup time of 0.2 ns. Then the cycle time is $T_c = 0.3 + 3 + 2 + 4 + 0.2 = 9.5$ ns. The circuit has a latency of 9.5 ns and a throughput of 1/9.5 ns = 105 MHz.

Figure 3.59 shows the same circuit partitioned into a two-stage pipeline by adding a register between blocks 3 and 4. The first stage has a minimum clock period of $0.3 + 3 + 2 + 0.2 = 5.5$ ns. The second stage has a minimum clock period of $0.3 + 4 + 0.2 = 4.5$ ns. The clock must be slow enough for all stages to work. Hence, $T_c = 5.5$ ns. The latency

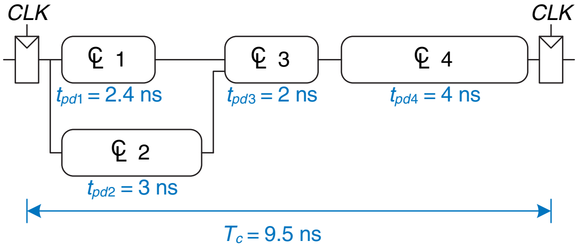


Figure 3.58 **Circuit with no pipelining**

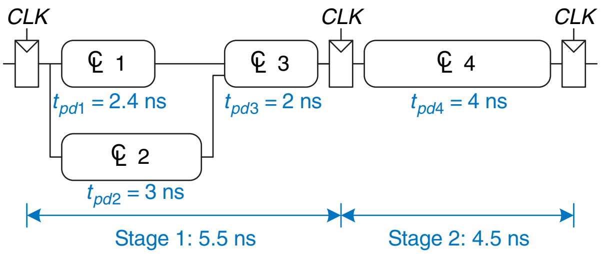

CLK
CLK
CLK
C L 1
C L 3
C L 4
C L 2
t_{pd1} = 2.4 ns
t_{pd3} = 2 ns
t_{pd4} = 4 ns
t_{pd2} = 3 ns
Stage 1: 5.5 ns
Stage 2: 4.5 ns

Figure 3.59 **Circuit with two-stage pipeline**

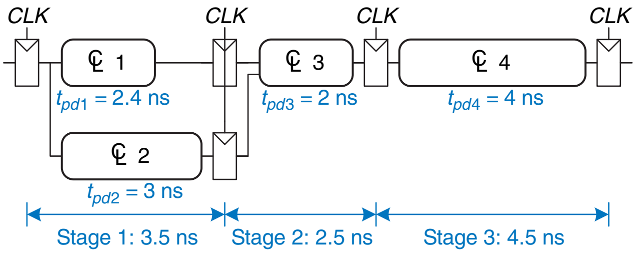

CLK
CLK
CLK
CLK
C L 1
C L 3
C L 4
C L 2
t_{pd1} = 2.4 ns
t_{pd3} = 2 ns
t_{pd4} = 4 ns
t_{pd2} = 3 ns
Stage 1: 3.5 ns
Stage 2: 2.5 ns
Stage 3: 4.5 ns

Figure 3.60 **Circuit with three-stage pipeline**

is two clock cycles, or 11 ns. The throughput is 1/5.5 ns = 182 MHz. This example shows that, in a real circuit, pipelining with two stages almost doubles the throughput and slightly increases the latency. In comparison, ideal pipelining would exactly double the throughput at no penalty in latency. The discrepancy comes about because the circuit cannot be divided into two exactly equal halves and because the registers introduce more sequencing overhead.

Figure 3.60 shows the same circuit partitioned into a three-stage pipeline. Note that two more registers are needed to store the results of blocks 1 and 2 at the end of the first pipeline stage. The cycle time is now limited by the third stage to 4.5 ns. The latency is three cycles, or 13.5 ns. The throughput is 1/4.5 ns = 222 MHz. Again, adding a pipeline stage improves throughput at the expense of some latency.

Although these techniques are powerful, they do not apply to all situations. The bane of parallelism is *dependencies*. If a current task is dependent on the result of a prior task, rather than just prior steps in the current task, the task cannot start until the prior task has completed. For example, if Ben wants to check that the first tray of cookies tastes good before he starts preparing the second, he has a dependency that prevents pipelining or parallel operation. Parallelism is one of the most important techniques for designing high-performance digital systems. Chapter 7 discusses pipelining further and shows examples of handling dependencies.

3.7 SUMMARY

This chapter has described the analysis and design of sequential logic. In contrast to combinational logic, whose outputs depend only on the current inputs, sequential logic outputs depend on both current and prior inputs. In other words, sequential logic remembers information about prior inputs. This memory is called the state of the logic.

Anyone who could invent logic whose outputs depend on future inputs would be fabulously wealthy!

Sequential circuits can be difficult to analyze and are easy to design incorrectly, so we limit ourselves to a small set of carefully designed building blocks. The most important element for our purposes is the flip-flop, which receives a clock and an input D and produces an output Q. The flip-flop copies D to Q on the rising edge of the clock and otherwise remembers the old state of Q. A group of flip-flops sharing a common clock is called a register. Flip-flops may also receive reset or enable control signals.

Although many forms of sequential logic exist, we discipline ourselves to use synchronous sequential circuits because they are easy to design. Synchronous sequential circuits consist of blocks of combinational logic separated by clocked registers. The state of the circuit is stored in the registers and updated only on clock edges.

Finite state machines are a powerful technique for designing sequential circuits. To design an FSM, first identify the inputs and outputs of the machine and sketch a state transition diagram, indicating the states and the transitions between them. Select an encoding for the states, and rewrite the diagram as a state transition table and output table, indicating the next state and output given the current state and input. From these tables, design the combinational logic to compute the next state and output, and sketch the circuit.

Synchronous sequential circuits have a timing specification including the clock-to-Q propagation and contamination delays, t_{pcq} and t_{ccq}, and the setup and hold times, t_{setup} and t_{hold}. For correct operation, their inputs must be stable during an aperture time that starts a setup time before the rising edge of the clock and ends a hold time after the rising edge of the clock. The minimum cycle time T_c of the system is equal to the propagation delay t_{pd} through the combinational logic plus $t_{pcq} + t_{\text{setup}}$ of the register. For correct operation, the contamination delay through the register and combinational logic must be greater than t_{hold}. Despite the common misconception to the contrary, hold time does not affect the cycle time.

Overall system performance is measured in latency and throughput. The latency is the time required for a token to pass from start to end. The throughput is the number of tokens that the system can process per unit time. Parallelism improves system throughput.

Exercises

Exercise 3.1 Given the input waveforms shown in Figure 3.61, sketch the output, Q, of an SR latch.

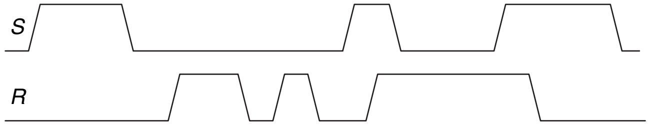


Figure 3.61 Input waveforms of SR latch for Exercise 3.1

Exercise 3.2 Given the input waveforms shown in Figure 3.62, sketch the output, Q, of an SR latch.

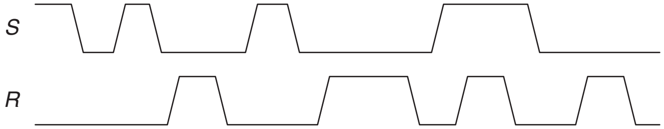


Figure 3.62 Input waveforms of SR latch for Exercise 3.2

Exercise 3.3 Given the input waveforms shown in Figure 3.63, sketch the output, Q, of a D latch.

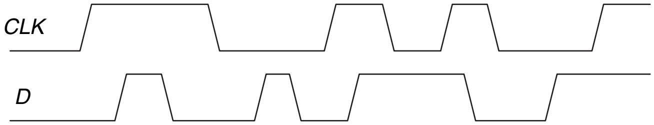


Figure 3.63 Input waveforms of D latch or flip-flop for Exercises 3.3 and 3.5

Exercise 3.4 Given the input waveforms shown in Figure 3.64, sketch the output, Q, of a D latch.

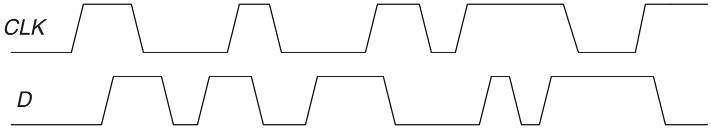


Figure 3.64 Input waveforms of D latch or flip-flop for Exercises 3.4 and 3.6

Exercise 3.5 Given the input waveforms shown in Figure 3.63, sketch the output, *Q*, of a D flip-flop.

Exercise 3.6 Given the input waveforms shown in Figure 3.64, sketch the output, *Q*, of a D flip-flop.

Exercise 3.7 Is the circuit in Figure 3.65 combinational logic or sequential logic? Explain in a simple fashion what the relationship is between the inputs and outputs. What would you call this circuit?

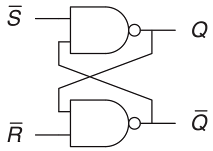


Figure 3.65 **Mystery circuit**

Exercise 3.8 Is the circuit in Figure 3.66 combinational logic or sequential logic? Explain in a simple fashion what the relationship is between the inputs and outputs. What would you call this circuit?

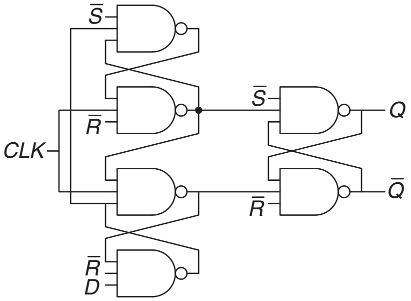


Figure 3.66 **Mystery circuit**

Exercise 3.9 The *toggle (T) flip-flop* has one input, *CLK*, and one output, *Q*. On each rising edge of *CLK*, *Q* toggles to the complement of its previous value. Draw a schematic for a T flip-flop using a D flip-flop and an inverter.

Exercise 3.10 *A JK flip-flop* receives a clock and two inputs, *J* and *K*. On the rising edge of the clock, it updates the output, *Q*. If *J* and *K* are both 0, *Q* retains its old value. If only *J* is 1, *Q* becomes 1. If only *K* is 1, *Q* becomes 0. If both *J* and *K* are 1, *Q* becomes the opposite of its present state.

(a) Construct a JK flip-flop using a D flip-flop and some combinational logic.

(b) Construct a D flip-flop using a JK flip-flop and some combinational logic.

(c) Construct a T flip-flop (see Exercise 3.9) using a JK flip-flop.

Exercise 3.11 The circuit in Figure 3.67 is called a *Muller C-element*. Explain in a simple fashion what the relationship is between the inputs and output.

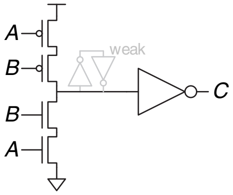


Figure 3.67 **Muller C-element**

Exercise 3.12 Design an asynchronously resettable D latch using logic gates.

Exercise 3.13 Design an asynchronously resettable D flip-flop using logic gates.

Exercise 3.14 Design a synchronously settable D flip-flop using logic gates.

Exercise 3.15 Design an asynchronously settable D flip-flop using logic gates.

Exercise 3.16 Suppose a ring oscillator is built from N inverters connected in a loop. Each inverter has a minimum delay of t_{cd} and a maximum delay of t_{pd}. If N is odd, determine the range of frequencies at which the oscillator might operate.

Exercise 3.17 Why must N be odd in Exercise 3.16?

Exercise 3.18 Which of the circuits in Figure 3.68 are synchronous sequential circuits? Explain.

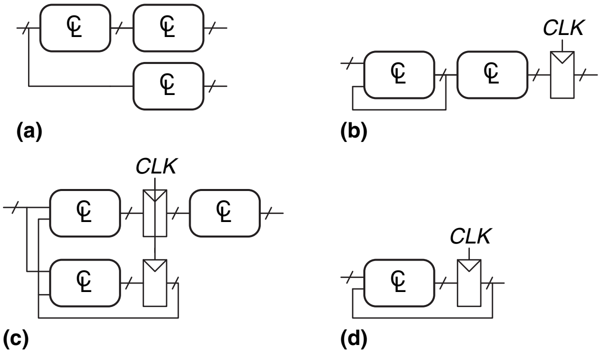


Figure 3.68 **Circuits**

Exercise 3.19 You are designing an elevator controller for a building with 25 floors. The controller has two inputs: *UP* and *DOWN*. It produces an output indicating the floor that the elevator is on. There is no floor 13. What is the minimum number of bits of state in the controller?

Exercise 3.20 You are designing an FSM to keep track of the mood of four students working in the digital design lab. Each student's mood is either HAPPY (the circuit works), SAD (the circuit blew up), BUSY (working on the circuit), CLUELESS (confused about the circuit), or ASLEEP (face down on the circuit board). How many states does the FSM have? What is the minimum number of bits necessary to represent these states?

Exercise 3.21 How would you factor the FSM from Exercise 3.20 into multiple simpler machines? How many states does each simpler machine have? What is the minimum total number of bits necessary in this factored design?

Exercise 3.22 Describe in words what the state machine in Figure 3.69 does. Using binary state encodings, complete a state transition table and output table for the FSM. Write Boolean equations for the next state and output and sketch a schematic of the FSM.

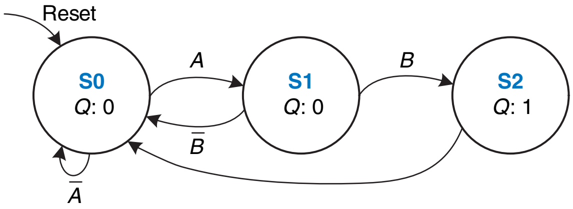


Figure 3.69 State transition diagram

Exercise 3.23 Describe in words what the state machine in Figure 3.70 does. Using binary state encodings, complete a state transition table and output table for the FSM. Write Boolean equations for the next state and output and sketch a schematic of the FSM.

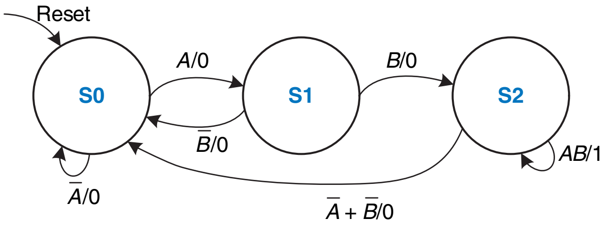


Figure 3.70 State transition diagram

Exercise 3.24 Accidents are still occurring at the intersection of Academic Avenue and Bravado Boulevard. The football team is rushing into the intersection the moment light *B* turns green. They are colliding with sleep-deprived CS majors who stagger into the intersection just before light *A* turns red. Extend the traffic light controller from Section 3.4.1 so that both lights are red for 5 seconds before either light turns green again. Sketch your improved Moore machine state transition diagram, state encodings, state transition table, output table, next state and output equations, and your FSM schematic.

Exercise 3.25 Alyssa P. Hacker's snail from Section 3.4.3 has a daughter with a Mealy machine FSM brain. The daughter snail smiles whenever she slides over the pattern 1101 or the pattern 1110. Sketch the state transition diagram for this happy snail using as few states as possible. Choose state encodings and write a combined state transition and output table using your encodings. Write the next state and output equations and sketch your FSM schematic.

Exercise 3.26 You have been enlisted to design a soda machine dispenser for your department lounge. Sodas are partially subsidized by the student chapter of the IEEE, so they cost only 25 cents. The machine accepts nickels, dimes, and quarters. When enough coins have been inserted, it dispenses the soda and returns any necessary change. Design an FSM controller for the soda machine. The FSM inputs are *Nickel*, *Dime*, and *Quarter*, indicating which coin was inserted. Assume that exactly one coin is inserted on each cycle. The outputs are *Dispense*, *ReturnNickel*, *ReturnDime*, and *ReturnTwoDimes*. When the FSM reaches 25 cents, it asserts *Dispense* and the necessary *Return* outputs required to deliver the appropriate change. Then it should be ready to start accepting coins for another soda.

Exercise 3.27 Gray codes have a useful property in that consecutive numbers differ in only a single bit position. Table 3.23 lists a 3-bit Gray code representing the numbers 0 to 7. Design a 3-bit modulo 8 Gray code counter FSM with no inputs and three outputs. (A modulo N counter counts from 0 to $N - 1$, then

Table 3.23 **3-bit Gray code**

Number	Gray code		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

repeats. For example, a watch uses a modulo 60 counter for the minutes and seconds that counts from 0 to 59.) When reset, the output should be 000. On each clock edge, the output should advance to the next Gray code. After reaching 100, it should repeat with 000.

Exercise 3.28 Extend your modulo 8 Gray code counter from Exercise 3.27 to be an UP/DOWN counter by adding an *UP* input. If $UP = 1$, the counter advances to the next number. If $UP = 0$, the counter retreats to the previous number.

Exercise 3.29 Your company, Detect-o-rama, would like to design an FSM that takes two inputs, *A* and *B*, and generates one output, *Z*. The output in cycle *n*, Z_n, is either the Boolean AND or OR of the corresponding input A_n and the previous input $A_{n\text{-}1}$, depending on the other input, B_n:

$$\begin{aligned} Z_n &= A_n A_{n-1} && \text{if } B_n = 0 \\ Z_n &= A_n + A_{n-1} && \text{if } B_n = 1 \end{aligned}$$

(a) Sketch the waveform for *Z* given the inputs shown in Figure 3.71.

(b) Is this FSM a Moore or a Mealy machine?

(c) Design the FSM. Show your state transition diagram, encoded state transition table, next state and output equations, and schematic.

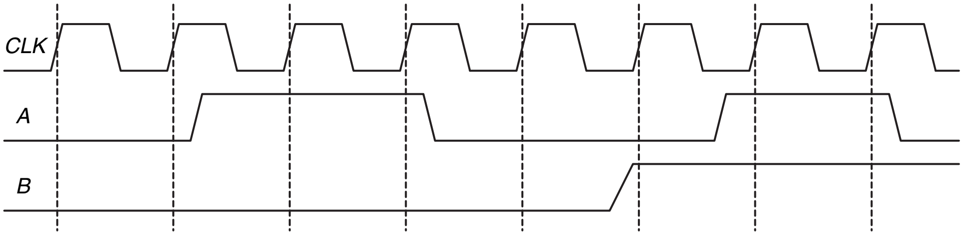


Figure 3.71 FSM input waveforms

Exercise 3.30 Design an FSM with one input, *A*, and two outputs, *X* and *Y*. *X* should be 1 if *A* has been 1 for at least three cycles altogether (not necessarily consecutively). *Y* should be 1 if *A* has been 1 for at least two consecutive cycles. Show your state transition diagram, encoded state transition table, next state and output equations, and schematic.

Exercise 3.31 Analyze the FSM shown in Figure 3.72. Write the state transition and output tables and sketch the state transition diagram. Describe in words what the FSM does.

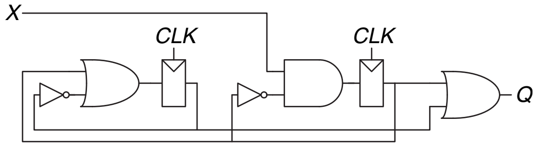


Figure 3.72 **FSM schematic**

Exercise 3.32 Repeat Exercise 3.31 for the FSM shown in Figure 3.73. Recall that the *s* and *r* register inputs indicate set and reset, respectively.

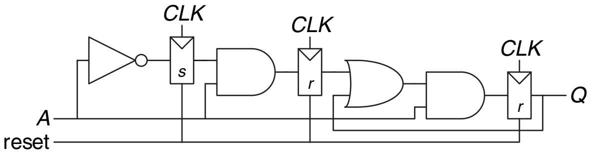


Figure 3.73 **FSM schematic**

Exercise 3.33 Ben Bitdiddle has designed the circuit in Figure 3.74 to compute a registered four-input XOR function. Each two-input XOR gate has a propagation delay of 100 ps and a contamination delay of 55 ps. Each flip-flop has a setup time of 60 ps, a hold time of 20 ps, a clock-to-Q maximum delay of 70 ps, and a clock-to-Q minimum delay of 50 ps.

(a) If there is no clock skew, what is the maximum operating frequency of the circuit?

(b) How much clock skew can the circuit tolerate if it must operate at 2 GHz?

(c) How much clock skew can the circuit tolerate before it might experience a hold time violation?

(d) Alyssa P. Hacker points out that she can redesign the combinational logic between the registers to be faster *and* tolerate more clock skew. Her improved circuit also uses three two-input XORs, but they are arranged differently. What is her circuit? What is its maximum frequency if there is no clock skew? How much clock skew can the circuit tolerate before it might experience a hold time violation?

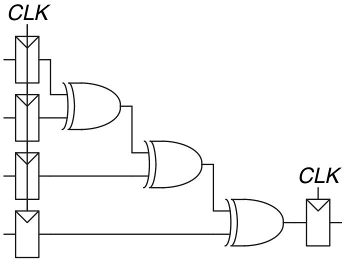


Figure 3.74 **Registered four-input XOR circuit**

Exercise 3.34 You are designing an adder for the blindingly fast 2-bit RePentium Processor. The adder is built from two full adders such that the carry out of the first adder is the carry in to the second adder, as shown in Figure 3.75. Your adder has input and output registers and must complete the addition in one clock cycle. Each full adder has the following propagation delays: 20 ps from C_{in} to C_{out} or to *Sum* (*S*), 25 ps from *A* or *B* to C_{out}, and 30 ps from *A* or *B* to *S*. The adder has a contamination delay of 15 ps from C_{in} to either output and 22 ps from *A* or *B* to either output. Each flip-flop has a setup time of 30 ps, a hold time of 10 ps, a clock-to-*Q* propagation delay of 35 ps, and a clock-to-*Q* contamination delay of 21 ps.

(a) If there is no clock skew, what is the maximum operating frequency of the circuit?

(b) How much clock skew can the circuit tolerate if it must operate at 8 GHz?

(c) How much clock skew can the circuit tolerate before it might experience a hold time violation?

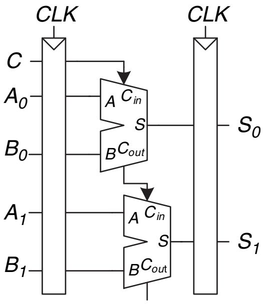


Figure 3.75 **2-bit adder schematic**

Exercise 3.35 A *field programmable gate array* (*FPGA*) uses *configurable logic blocks* (*CLBs*) rather than logic gates to implement combinational logic. The Xilinx Spartan 3 FPGA has propagation and contamination delays of 0.61 and 0.30 ns, respectively, for each CLB. It also contains flip-flops with propagation and contamination delays of 0.72 and 0.50 ns, and setup and hold times of 0.53 and 0 ns, respectively.

(a) If you are building a system that needs to run at 40 MHz, how many consecutive CLBs can you use between two flip-flops? Assume there is no clock skew and no delay through wires between CLBs.

(b) Suppose that all paths between flip-flops pass through at least one CLB. How much clock skew can the FPGA have without violating the hold time?

Exercise 3.36 A synchronizer is built from a pair of flip-flops with $t_{\text{setup}} = 50$ ps, $T_0 = 20$ ps, and $\tau = 30$ ps. It samples an asynchronous input that changes 10^8 times per second. What is the minimum clock period of the synchronizer to achieve a mean time between failures (MTBF) of 100 years?

Exercise 3.37 You would like to build a synchronizer that can receive asynchronous inputs with an MTBF of 50 years. Your system is running at 1 GHz, and you use sampling flip-flops with $\tau = 100$ ps, $T_0 = 110$ ps, and $t_{setup} = 70$ ps. The synchronizer receives a new asynchronous input on average 0.5 times per second (i.e., once every 2 seconds). What is the required probability of failure to satisfy this MTBF? How many clock cycles would you have to wait before reading the sampled input signal to give that probability of error?

Exercise 3.38 You are walking down the hallway when you run into your lab partner walking in the other direction. The two of you first step one way and are still in each other's way. Then you both step the other way and are still in each other's way. Then you both wait a bit, hoping the other person will step aside. You can model this situation as a metastable point and apply the same theory that has been applied to synchronizers and flip-flops. Suppose you create a mathematical model for yourself and your lab partner. You start the unfortunate encounter in the metastable state. The probability that you remain in this state after t seconds is $e^{-\frac{t}{\tau}}$. τ indicates your response rate; today, your brain has been blurred by lack of sleep and has $\tau = 20$ seconds.

(a) How long will it be until you have 99% certainty that you will have resolved from metastability (i.e., figured out how to pass one another)?

(b) You are not only sleepy, but also ravenously hungry. In fact, you will starve to death if you don't get going to the cafeteria within 3 minutes. What is the probability that your lab partner will have to drag you to the morgue?

Exercise 3.39 You have built a synchronizer using flip-flops with $T_0 = 20$ ps and $\tau = 30$ ps. Your boss tells you that you need to increase the MTBF by a factor of 10. By how much do you need to increase the clock period?

Exercise 3.40 Ben Bitdiddle invents a new and improved synchronizer in Figure 3.76 that he claims eliminates metastability in a single cycle. He explains that the circuit in box M is an analog "metastability detector" that produces a HIGH output if the input voltage is in the forbidden zone between V_{IL} and V_{IH}. The metastability detector checks to determine whether the first flip-flop has produced a metastable output on $D2$. If so, it asynchronously resets the flip-flop to produce a good 0 at $D2$. The second flip-flop then samples $D2$, always producing a valid logic level on Q. Alyssa P. Hacker tells Ben that there must be a bug in the circuit, because eliminating metastability is just as impossible as building a perpetual motion machine. Who is right? Explain, showing Ben's error or showing why Alyssa is wrong.

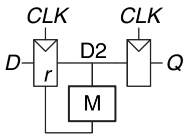


Figure 3.76 **"New and improved" synchronizer**

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 3.1 Draw a state machine that can detect when it has received the serial input sequence 01010.

Question 3.2 Design a serial (one bit at a time) two's complementer FSM with two inputs, *Start* and A, and one output, Q. A binary number of arbitrary length is provided to input A, starting with the least significant bit. The corresponding bit of the output appears at Q on the same cycle. *Start* is asserted for one cycle to initialize the FSM before the least significant bit is provided.

Question 3.3 What is the difference between a latch and a flip-flop? Under what circumstances is each one preferable?

Question 3.4 Design a 5-bit counter finite state machine.

Question 3.5 Design an edge detector circuit. The output should go HIGH for one cycle after the input makes a 0 → 1 transition.

Question 3.6 Describe the concept of pipelining and why it is used.

Question 3.7 Describe what it means for a flip-flop to have a negative hold time.

Question 3.8 Given signal A, shown in Figure 3.77, design a circuit that produces signal B.

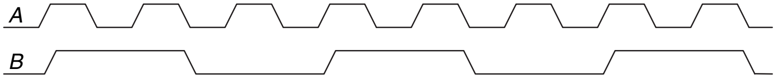
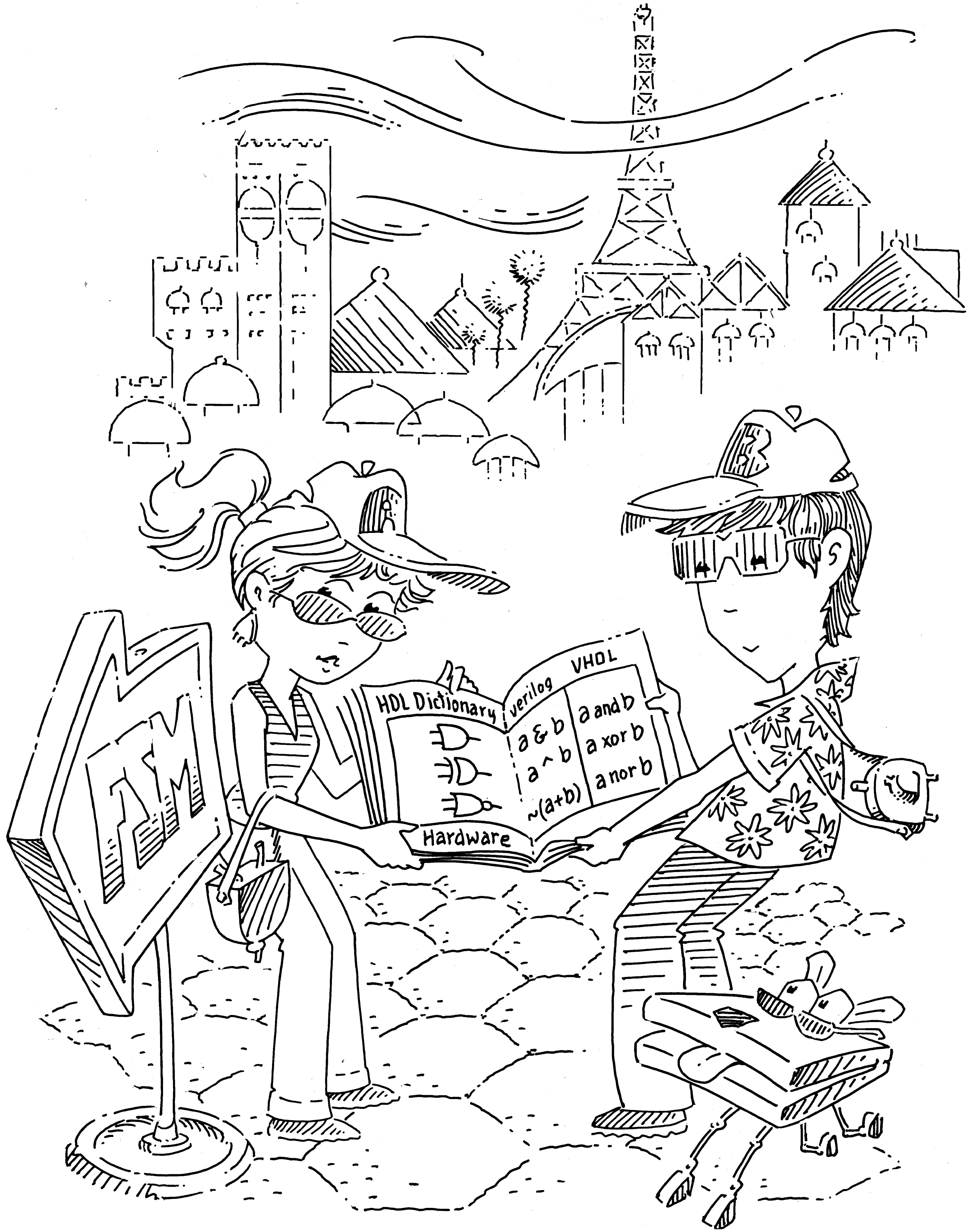


Figure 3.77 **Signal waveforms**

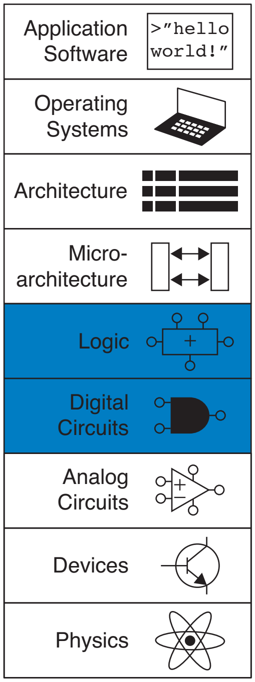
Question 3.9 Consider a block of logic between two registers. Explain the timing constraints. If you add a buffer on the clock input of the receiver (the second flip-flop), does the setup time constraint get better or worse?


FSM
HDL Dictionary
verilog
VHDL
a & b
a and b
a ^ b
a xor b
~(a+b)
a nor b
Hardware

Hardware Description Languages

4

4.1 Introduction
4.2 Combinational Logic
4.3 Structural Modeling
4.4 Sequential Logic
4.5 More Combinational Logic
4.6 Finite State Machines
4.7 Data Types*
4.8 Parameterized Modules*
4.9 Testbenches
4.10 Summary
Exercises
Interview Questions



4.1 INTRODUCTION

Thus far, we have focused on designing combinational and sequential digital circuits at the schematic level. The process of finding an efficient set of logic gates to perform a given function is labor intensive and error prone, requiring manual simplification of truth tables or Boolean equations and manual translation of finite state machines (FSMs) into gates. In the 1990s, designers discovered that they were far more productive if they worked at a higher level of abstraction, specifying just the logical function and allowing a *computer-aided design* (*CAD*) tool to produce the optimized gates. The specifications are generally given in a *hardware description language* (*HDL*). The two leading hardware description languages are *SystemVerilog* and *VHDL*.

SystemVerilog and VHDL are built on similar principles but have different syntax. Discussion of these languages in this chapter is divided into two columns for literal side-by-side comparison, with SystemVerilog on the left and VHDL on the right. When you read the chapter for the first time, focus on one language or the other. Once you know one, you'll quickly master the other if you need it.

Subsequent chapters show hardware in both schematic and HDL form. If you choose to skip this chapter and not learn one of the HDLs, you will still be able to master the principles of computer organization from the schematics. However, the vast majority of commercial systems are now built using HDLs rather than schematics. If you expect to do digital design at any point in your professional life, we urge you to learn one of the HDLs.

4.1.1 Modules

A block of hardware with inputs and outputs is called a *module*. An AND gate, a multiplexer, and a priority circuit are all examples of hardware modules. The two general styles for describing module functionality are

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00004-5
© 2013 Elsevier, Inc. All rights reserved.

behavioral and *structural*. Behavioral models describe what a module does. Structural models describe how a module is built from simpler pieces; it is an application of hierarchy. The SystemVerilog and VHDL code in HDL Example 4.1 illustrate behavioral descriptions of a module that computes the Boolean function from Example 2.6, $y = \overline{a}\,\overline{b}\,\overline{c} + a\overline{b}\,\overline{c} + a\overline{b}c$. In both languages, the module is named `sillyfunction` and has three inputs, `a`, `b`, and `c`, and one output, `y`.

HDL Example 4.1 COMBINATIONAL LOGIC

SystemVerilog

```
module sillyfunction(input  logic a, b, c,
                     output logic y);

  assign y = ~a & ~b & ~c |
              a & ~b & ~c |
              a & ~b &  c;

endmodule
```

A SystemVerilog module begins with the module name and a listing of the inputs and outputs. The `assign` statement describes combinational logic. `~` indicates NOT, `&` indicates AND, and `|` indicates OR.

`logic` signals such as the inputs and outputs are Boolean variables (0 or 1). They may also have floating and undefined values, as discussed in Section 4.2.8.

The `logic` type was introduced in SystemVerilog. It supersedes the `reg` type, which was a perennial source of confusion in Verilog. `logic` should be used everywhere except on signals with multiple drivers. Signals with multiple drivers are called *nets* and will be explained in Section 4.7.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sillyfunction is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of sillyfunction is
begin
  y <= (not a and not b and not c) or
       (a and not b and not c) or
       (a and not b and c);
end;
```

VHDL code has three parts: the `library` use clause, the `entity` declaration, and the `architecture` body. The `library` use clause will be discussed in Section 4.7.2. The `entity` declaration lists the module name and its inputs and outputs. The `architecture` body defines what the module does.

VHDL signals, such as inputs and outputs, must have a *type declaration*. Digital signals should be declared to be `STD_LOGIC` type. `STD_LOGIC` signals can have a value of `'0'` or `'1'`, as well as floating and undefined values that will be described in Section 4.2.8. The `STD_LOGIC` type is defined in the `IEEE.STD_LOGIC_1164` library, which is why the library must be used.

VHDL lacks a good default order of operations between AND and OR, so Boolean equations should be parenthesized.

A module, as you might expect, is a good application of modularity. It has a well defined interface, consisting of its inputs and outputs, and it performs a specific function. The particular way in which it is coded is unimportant to others that might use the module, as long as it performs its function.

4.1.2 Language Origins

Universities are almost evenly split on which of these languages is taught in a first course. Industry is trending toward SystemVerilog, but many companies still use VHDL and many designers need to be fluent in both.

SystemVerilog

Verilog was developed by Gateway Design Automation as a proprietary language for logic simulation in 1984. Gateway was acquired by Cadence in 1989 and Verilog was made an open standard in 1990 under the control of Open Verilog International. The language became an IEEE standard[1] in 1995. The language was extended in 2005 to streamline idiosyncracies and to better support modeling and verification of systems. These extensions have been merged into a single language standard, which is now called SystemVerilog (IEEE STD 1800-2009). SystemVerilog file names normally end in `.sv`.

VHDL

VHDL is an acronym for the *VHSIC Hardware Description Language*. VHSIC is in turn an acronym for the *Very High Speed Integrated Circuits* program of the US Department of Defense.

VHDL was originally developed in 1981 by the Department of Defense to describe the structure and function of hardware. Its roots draw from the Ada programming language. The language was first envisioned for documentation but was quickly adopted for simulation and synthesis. The IEEE standardized it in 1987 and has updated the standard several times since. This chapter is based on the 2008 revision of the VHDL standard (IEEE STD 1076-2008), which streamlines the language in a variety of ways. At the time of this writing, not all of the VHDL 2008 features are supported by CAD tools; this chapter only uses those understood by Synplicity, Altera Quartus, and ModelSim. VHDL file names normally end in `.vhd`.

Compared to SystemVerilog, VHDL is more verbose and cumbersome, as you might expect of a language developed by committee.

Both languages are fully capable of describing any hardware system, and both have their quirks. The best language to use is the one that is already being used at your site or the one that your customers demand. Most CAD tools today allow the two languages to be mixed, so that different modules can be described in different languages.

4.1.3 Simulation and Synthesis

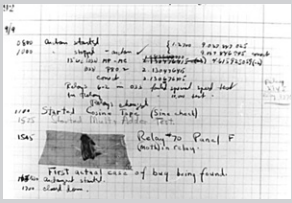
The two major purposes of HDLs are logic *simulation* and *synthesis*. During simulation, inputs are applied to a module, and the outputs are checked to verify that the module operates correctly. During synthesis, the textual description of a module is transformed into logic gates.

Simulation

Humans routinely make mistakes. Such errors in hardware designs are called *bugs*. Eliminating the bugs from a digital system is obviously important, especially when customers are paying money and lives depend on the correct operation. Testing a system in the laboratory is time-consuming. Discovering the cause of errors in the lab can be extremely difficult, because only signals routed to the chip pins can be observed. There is no way to directly observe what is happening inside a chip. Correcting errors after the system is built can be devastatingly expensive. For example,

The term "bug" predates the invention of the computer. Thomas Edison called the "little faults and difficulties" with his inventions "bugs" in 1878.

The first real computer bug was a moth, which got caught between the relays of the Harvard Mark II electromechanical computer in 1947. It was found by Grace Hopper, who logged the incident, along with the moth itself and the comment "first actual case of bug being found."



Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.
1545 Relay #70 Panel F (moth) in relay.
First actual case of bug being found.

Source: Notebook entry courtesy Naval Historical Center, US Navy; photo No. NII 96566-KN)

---

[1] The Institute of Electrical and Electronics Engineers (IEEE) is a professional society responsible for many computing standards including Wi-Fi (802.11), Ethernet (802.3), and floating-point numbers (754).

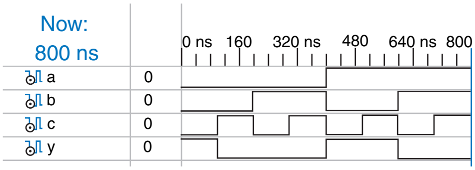


Figure 4.1 Simulation waveforms

correcting a mistake in a cutting-edge integrated circuit costs more than a million dollars and takes several months. Intel's infamous FDIV (floating point division) bug in the Pentium processor forced the company to recall chips after they had shipped, at a total cost of $475 million. Logic simulation is essential to test a system before it is built.

Figure 4.1 shows waveforms from a simulation[2] of the previous `sillyfunction` module demonstrating that the module works correctly. `y` is TRUE when `a`, `b`, and `c` are 000, 100, or 101, as specified by the Boolean equation.

Synthesis

Logic synthesis transforms HDL code into a *netlist* describing the hardware (e.g., the logic gates and the wires connecting them). The logic synthesizer might perform optimizations to reduce the amount of hardware required. The netlist may be a text file, or it may be drawn as a schematic to help visualize the circuit. Figure 4.2 shows the results of synthesizing the `sillyfunction` module.[3] Notice how the three three-input AND gates are simplified into two two-input AND gates, as we discovered in Example 2.6 using Boolean algebra.

Circuit descriptions in HDL resemble code in a programming language. However, you must remember that the code is intended to represent hardware. SystemVerilog and VHDL are rich languages with many commands. Not all of these commands can be synthesized into hardware.

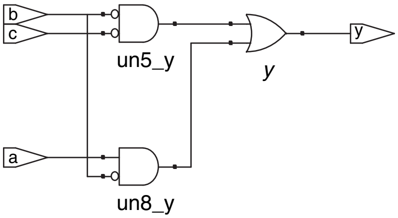


Figure 4.2 Synthesized circuit

---

[2] The simulation was performed with the ModelSim PE Student Edition Version 10.0c. ModelSim was selected because it is used commercially, yet a student version with a capacity of 10,000 lines of code is freely available.

[3] Synthesis was performed with Synplify Premier from Synplicity. The tool was selected because it is the leading commercial tool for synthesizing HDL to field-programmable gate arrays (see Section 5.6.2) and because it is available inexpensively for universities.

For example, a command to print results on the screen during simulation does not translate into hardware. Because our primary interest is to build hardware, we will emphasize a *synthesizable subset* of the languages. Specifically, we will divide HDL code into *synthesizable* modules and a *testbench*. The synthesizable modules describe the hardware. The testbench contains code to apply inputs to a module, check whether the output results are correct, and print discrepancies between expected and actual outputs. Testbench code is intended only for simulation and cannot be synthesized.

One of the most common mistakes for beginners is to think of HDL as a computer program rather than as a shorthand for describing digital hardware. If you don't know approximately what hardware your HDL should synthesize into, you probably won't like what you get. You might create far more hardware than is necessary, or you might write code that simulates correctly but cannot be implemented in hardware. Instead, think of your system in terms of blocks of combinational logic, registers, and finite state machines. Sketch these blocks on paper and show how they are connected before you start writing code.

In our experience, the best way to learn an HDL is by example. HDLs have specific ways of describing various classes of logic; these ways are called *idioms*. This chapter will teach you how to write the proper HDL idioms for each type of block and then how to put the blocks together to produce a working system. When you need to describe a particular kind of hardware, look for a similar example and adapt it to your purpose. We do not attempt to rigorously define all the syntax of the HDLs, because that is deathly boring and because it tends to encourage thinking of HDLs as programming languages, not shorthand for hardware. The IEEE SystemVerilog and VHDL specifications, and numerous dry but exhaustive textbooks, contain all of the details, should you find yourself needing more information on a particular topic. (See the Further Readings section at the back of the book.)

4.2 COMBINATIONAL LOGIC

Recall that we are disciplining ourselves to design synchronous sequential circuits, which consist of combinational logic and registers. The outputs of combinational logic depend only on the current inputs. This section describes how to write behavioral models of combinational logic with HDLs.

4.2.1 Bitwise Operators

Bitwise operators act on single-bit signals or on multi-bit busses. For example, the `inv` module in HDL Example 4.2 describes four inverters connected to 4-bit busses.

HDL Example 4.2 INVERTERS

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

`a[3:0]` represents a 4-bit bus. The bits, from most significant to least significant, are `a[3]`, `a[2]`, `a[1]`, and `a[0]`. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have named the bus `a[4:1]`, in which case `a[4]` would have been the most significant. Or we could have used `a[0:3]`, in which case the bits, from most significant to least significant, would be `a[0]`, `a[1]`, `a[2]`, and `a[3]`. This is called *big-endian* order.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

VHDL uses `STD_LOGIC_VECTOR` to indicate busses of `STD_LOGIC`. `STD_LOGIC_VECTOR(3 downto 0)` represents a 4-bit bus. The bits, from most significant to least significant, are `a(3)`, `a(2)`, `a(1)`, and `a(0)`. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have declared the bus to be `STD_LOGIC_VECTOR(4 downto 1)`, in which case bit 4 would have been the most significant. Or we could have written `STD_LOGIC_VECTOR(0 to 3)`, in which case the bits, from most significant to least significant, would be `a(0)`, `a(1)`, `a(2)`, and `a(3)`. This is called *big-endian* order.

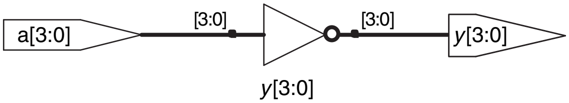


Figure 4.3 `inv` **synthesized circuit**

The endianness of a bus is purely arbitrary. (See the sidebar in Section 6.2.2 for the origin of the term.) Indeed, endianness is also irrelevant to this example, because a bank of inverters doesn't care what the order of the bits are. Endianness matters only for operators, such as addition, where the sum of one column carries over into the next. Either ordering is acceptable, as long as it is used consistently. We will consistently use the little-endian order, `[N – 1:0]` in SystemVerilog and `(N – 1 downto 0)` in VHDL, for an N-bit bus.

After each code example in this chapter is a schematic produced from the SystemVerilog code by the Synplify Premier synthesis tool. Figure 4.3 shows that the `inv` module synthesizes to a bank of four inverters, indicated by the inverter symbol labeled `y[3:0]`. The bank of inverters connects to 4-bit input and output busses. Similar hardware is produced from the synthesized VHDL code.

The `gates` module in HDL Example 4.3 demonstrates bitwise operations acting on 4-bit busses for other basic logic functions.

HDL Example 4.3 LOGIC GATES

SystemVerilog

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2,
                                y3, y4, y5);

  /* five different two-input logic
     gates acting on 4-bit busses */
  assign y1 = a & b;     // AND
  assign y2 = a | b;     // OR
  assign y3 = a ^ b;     // XOR
  assign y4 = ~(a & b);  // NAND
  assign y5 = ~(a | b);  // NOR
endmodule
```

~, ^, and | are examples of SystemVerilog *operators*, whereas `a`, `b`, and `y1` are *operands*. A combination of operators and operands, such as `a & b`, or `~(a | b)`, is called an *expression*. A complete command such as `assign y4 = ~(a & b);` is called a *statement*.

`assign out = in1 op in2;` is called a *continuous assignment statement*. Continuous assignment statements end with a semicolon. Anytime the inputs on the right side of the = in a continuous assignment statement change, the output on the left side is recomputed. Thus, continuous assignment statements describe combinational logic.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
     y1, y2, y3, y4,
     y5:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
  -- five different two-input logic gates
  -- acting on 4-bit busses
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

`not`, `xor`, and `or` are examples of VHDL *operators*, whereas `a`, `b`, and `y1` are *operands*. A combination of operators and operands, such as `a and b`, or `a nor b`, is called an *expression*. A complete command such as `y4 <= a nand b;` is called a *statement*.

`out <= in1 op in2;` is called a *concurrent signal assignment statement*. VHDL assignment statements end with a semicolon. Anytime the inputs on the right side of the `<=` in a concurrent signal assignment statement change, the output on the left side is recomputed. Thus, concurrent signal assignment statements describe combinational logic.

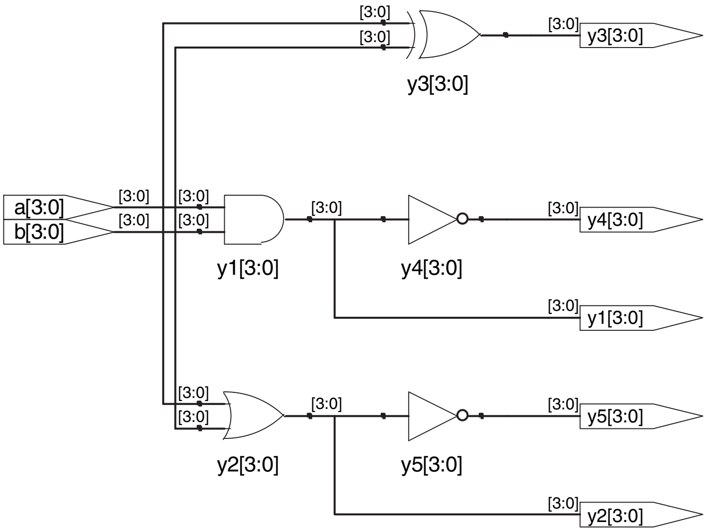


Figure 4.4 `gates` **synthesized circuit**

4.2.2 Comments and White Space

The `gates` example showed how to format comments. SystemVerilog and VHDL are not picky about the use of white space (i.e., spaces, tabs, and line breaks). Nevertheless, proper indenting and use of blank lines is helpful to make nontrivial designs readable. Be consistent in your use of capitalization and underscores in signal and module names. This text uses all lower case. Module and signal names must not begin with a digit.

SystemVerilog

SystemVerilog comments are just like those in C or Java. Comments beginning with `/*` continue, possibly across multiple lines, to the next `*/`. Comments beginning with `//` continue to the end of the line.

SystemVerilog is case-sensitive. `y1` and `Y1` are different signals in SystemVerilog. However, it is confusing to use multiple signals that differ only in case.

VHDL

Comments beginning with `/*` continue, possibly across multiple lines, to the next `*/`. Comments beginning with `--` continue to the end of the line.

VHDL is not case-sensitive. `y1` and `Y1` are the same signal in VHDL. However, other tools that may read your file might be case sensitive, leading to nasty bugs if you blithely mix upper and lower case.

4.2.3 Reduction Operators

Reduction operators imply a multiple-input gate acting on a single bus. HDL Example 4.4 describes an eight-input AND gate with inputs a_7, a_6, . . ., a_0. Analogous reduction operators exist for OR, XOR, NAND, NOR, and XNOR gates. Recall that a multiple-input XOR performs parity, returning TRUE if an odd number of inputs are TRUE.

HDL Example 4.4 EIGHT-INPUT AND

SystemVerilog

```
module and8(input  logic [7:0] a,
            output logic       y);

  assign y = &a;

  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and8 is
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of and8 is
begin
  y <= and a;
  -- and a is much easier to write than
  -- y <= a(7) and a(6) and a(5) and a(4) and
  --      a(3) and a(2) and a(1) and a(0);
end;
```

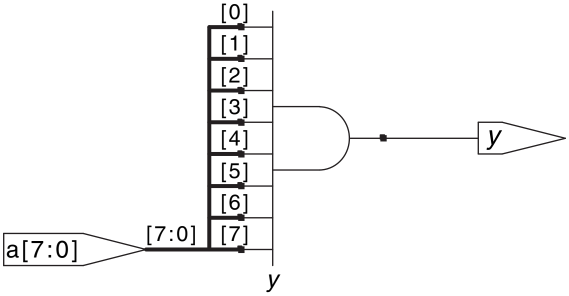


Figure 4.5 `and8` **synthesized circuit**

4.2.4 Conditional Assignment

Conditional assignments select the output from among alternatives based on an input called the *condition*. HDL Example 4.5 illustrates a 2:1 multiplexer using conditional assignment.

HDL Example 4.5 2:1 MULTIPLEXER

SystemVerilog

The *conditional operator* `?:` chooses, based on a first expression, between a second and third expression. The first expression is called the *condition*. If the condition is 1, the operator chooses the second expression. If the condition is 0, the operator chooses the third expression.

`?:` is especially useful for describing a multiplexer because, based on the first input, it selects between two others. The following code demonstrates the idiom for a 2:1 multiplexer with 4-bit inputs and outputs using the conditional operator.

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

  assign y = s ? d1 : d0;
endmodule
```

If `s` is 1, then `y = d1`. If `s` is 0, then `y = d0`.

`?:` is also called a *ternary operator*, because it takes three inputs. It is used for the same purpose in the C and Java programming languages.

VHDL

Conditional signal assignments perform different operations depending on some condition. They are especially useful for describing a multiplexer. For example, a 2:1 multiplexer can use conditional signal assignment to select one of two 4-bit inputs.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

The conditional signal assignment sets `y` to `d1` if `s` is 1. Otherwise it sets `y` to `d0`. Note that prior to the 2008 revision of VHDL, one had to write `when s = '1'` rather than `when s`.

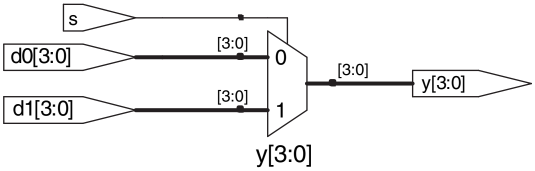


Figure 4.6 `mux2` **synthesized circuit**

HDL Example 4.6 shows a 4:1 multiplexer based on the same principle as the 2:1 multiplexer in HDL Example 4.5. Figure 4.7 shows the schematic for the 4:1 multiplexer produced by Synplify Premier. The software uses a different multiplexer symbol than this text has shown so far. The multiplexer has multiple data `(d)` and one-hot enable `(e)` inputs. When one of the enables is asserted, the associated data is passed to the output. For example, when `s[1] = s[0] = 0`, the bottom AND gate, `un1_s_5`, produces a 1, enabling the bottom input of the multiplexer and causing it to select `d0[3:0]`.

HDL Example 4.6 4:1 MULTIPLEXER

SystemVerilog

A 4:1 multiplexer can select one of four inputs using nested conditional operators.

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```

If `s[1]` is 1, then the multiplexer chooses the first expression, `(s[0] ? d3 : d2)`. This expression in turn chooses either `d3` or `d2` based on `s[0]` (y = `d3` if `s[0]` is 1 and `d2` if `s[0]` is 0). If `s[1]` is 0, then the multiplexer similarly chooses the second expression, which gives either `d1` or `d0` based on `s[0]`.

VHDL

A 4:1 multiplexer can select one of four inputs using multiple `else` clauses in the conditional signal assignment.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth1 of mux4 is
begin
  y <= d0 when s = "00" else
       d1 when s = "01" else
       d2 when s = "10" else
       d3;
end;
```

VHDL also supports *selected signal assignment statements* to provide a shorthand when selecting from one of several possibilities. This is analogous to using a `switch/case` statement in place of multiple `if/else` statements in some programming languages. The 4:1 multiplexer can be rewritten with selected signal assignment as follows:

```
architecture synth2 of mux4 is
begin
  with s select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
end;
```

4.2.5 Internal Variables

Often it is convenient to break a complex function into intermediate steps. For example, a full adder, which will be described in Section 5.2.1,

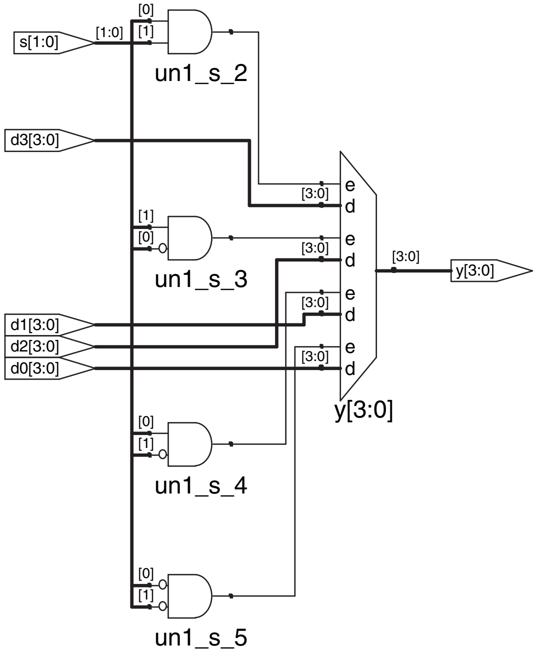


Figure 4.7 `mux4` **synthesized circuit**

is a circuit with three inputs and two outputs defined by the following equations:

$$\begin{aligned} S &= A \oplus B \oplus C_{\text{in}} \\ C_{\text{out}} &= AB + AC_{\text{in}} + BC_{\text{in}} \end{aligned} \tag{4.1}$$

If we define intermediate signals, P and G,

$$\begin{aligned} P &= A \oplus B \\ G &= AB \end{aligned} \tag{4.2}$$

we can rewrite the full adder as follows:

$$\begin{aligned} S &= P \oplus C_{\text{in}} \\ C_{\text{out}} &= G + PC_{\text{in}} \end{aligned} \tag{4.3}$$

Check this by filling out the truth table to convince yourself it is correct.

P and G are called *internal variables*, because they are neither inputs nor outputs but are used only internal to the module. They are similar to local variables in programming languages. HDL Example 4.7 shows how they are used in HDLs.

HDL assignment statements (`assign` in SystemVerilog and <= in VHDL) take place concurrently. This is different from conventional programming languages such as C or Java, in which statements are evaluated in the order in which they are written. In a conventional language, it is

HDL Example 4.7 FULL ADDER

SystemVerilog

In SystemVerilog, internal signals are usually declared as `logic`.

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

VHDL

In VHDL, *signals* are used to represent internal variables whose values are defined by *concurrent signal assignment statements* such as `p <= a xor b;`

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;

  s <= p xor cin;
  cout <= g or (p and cin);
end;
```

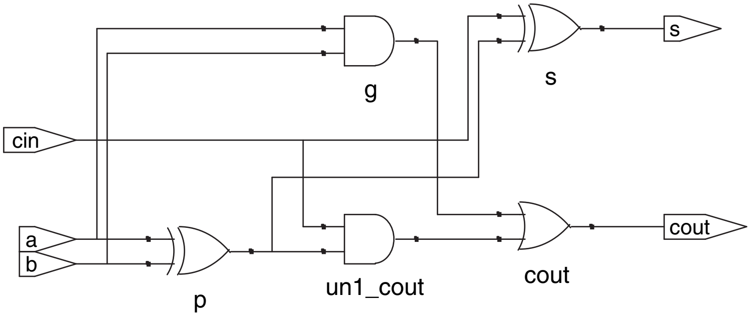


Figure 4.8 `fulladder` **synthesized circuit**

important that $S = P \oplus C_{in}$ comes after $P = A \oplus B$, because statements are executed sequentially. In an HDL, the order does not matter. Like hardware, HDL assignment statements are evaluated any time the inputs, signals on the right hand side, change their value, regardless of the order in which the assignment statements appear in a module.

4.2.6 Precedence

Notice that we parenthesized the `cout` computation in HDL Example 4.7 to define the order of operations as $C_{out} = G + (P \cdot C_{in})$, rather than $C_{out} = (G + P) \cdot C_{in}$. If we had not used parentheses, the default operation

HDL Example 4.8 OPERATOR PRECEDENCE

SystemVerilog

Table 4.1 SystemVerilog operator precedence

	Op	Meaning
Highest	`~`	NOT
	`*, /, %`	MUL, DIV, MOD
	`+, -`	PLUS, MINUS
	`<<, >>`	Logical Left/Right Shift
	`<<<, >>>`	Arithmetic Left/Right Shift
	`<, <=, >, >=`	Relative Comparison
	`==, !=`	Equality Comparison
Lowest	`&, ~&`	AND, NAND
	`^, ~^`	XOR, XNOR
	`|, ~|`	OR, NOR
	`?:`	Conditional

The operator precedence for SystemVerilog is much like you would expect in other programming languages. In particular, AND has precedence over OR. We could take advantage of this precedence to eliminate the parentheses.

```
assign cout = g | p & cin;
```

VHDL

Table 4.2 VHDL operator precedence

	Op	Meaning
Highest	`not`	NOT
	`*, /, mod, rem`	MUL, DIV, MOD, REM
	`+, -`	PLUS, MINUS
	`rol, ror,` `srl, sll`	Rotate, Shift logical
Lowest	`<, <=, >, >=`	Relative Comparison
	`=, /=`	Equality Comparison

	`and, or,`
	`nand, nor,`
	`xor, xnor`
		Logical Operations

Multiplication has precedence over addition in VHDL, as you would expect. However, unlike SystemVerilog, all of the logical operations (`and`, `or`, etc.) have equal precedence, unlike what one might expect in Boolean algebra. Thus, parentheses are necessary; otherwise `cout <= g or p and cin` would be interpreted from left to right as `cout <= (g or p) and cin`.

order is defined by the language. HDL Example 4.8 specifies operator precedence from highest to lowest for each language. The tables include arithmetic, shift, and comparison operators that will be defined in Chapter 5.

4.2.7 Numbers

Numbers can be specified in binary, octal, decimal, or hexadecimal (bases 2, 8, 10, and 16, respectively). The size, i.e., the number of bits, may optionally be given, and leading zeros are inserted to reach this size. Underscores in numbers are ignored and can be helpful in breaking long numbers into more readable chunks. HDL Example 4.9 explains how numbers are written in each language.

HDL Example 4.9 NUMBERS

SystemVerilog

The format for declaring constants is `N'Bvalue`, where `N` is the size in bits, `B` is a letter indicating the base, and `value` gives the value. For example, `9'h25` indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. SystemVerilog supports `'b` for binary, `'o` for octal, `'d` for decimal, and `'h` for hexadecimal. If the base is omitted, it defaults to decimal.

If the size is not given, the number is assumed to have as many bits as the expression in which it is being used. Zeros are automatically padded on the front of the number to bring it up to full size. For example, if `w` is a 6-bit bus, `assign w = 'b11` gives `w` the value 000011. It is better practice to explicitly give the size. An exception is that `'0` and `'1` are SystemVerilog idioms for filling a bus with all 0s and all 1s, respectively.

Table 4.3 **SystemVerilog numbers**

Numbers	Bits	Base	Val	Stored
`3'b101`	3	2	5	101
`'b11`	?	2	3	000 … 0011
`8'b11`	8	2	3	00000011
`8'b1010_1011`	8	2	171	10101011
`3'd6`	3	10	6	110
`6'o42`	6	8	34	100010
`8'hAB`	8	16	171	10101011
`42`	?	10	42	00 … 0101010

VHDL

In VHDL, STD_LOGIC numbers are written in binary and enclosed in single quotes: `'0'` and `'1'` indicate logic 0 and 1. The format for declaring STD_LOGIC_VECTOR constants is `NB"value"`, where `N` is the size in bits, `B` is a letter indicating the base, and `value` gives the value. For example, `9X"25"` indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. VHDL 2008 supports `B` for binary, `O` for octal, `D` for decimal, and `X` for hexadecimal.

If the base is omitted, it defaults to binary. If the size is not given, the number is assumed to have a size matching the number of bits specified in the value. As of October 2011, Synplify Premier from Synopsys does not yet support specifying the size.

`others => '0'` and `others => '1'` are VHDL idioms to fill all of the bits with 0 and 1, respectively.

Table 4.4 **VHDL numbers**

Numbers	Bits	Base	Val	Stored
`3B"101"`	3	2	5	101
`B"11"`	2	2	3	11
`8B"11"`	8	2	3	00000011
`8B"1010_1011"`	8	2	171	10101011
`3D"6"`	3	10	6	110
`6O"42"`	6	8	34	100010
`8X"AB"`	8	16	171	10101011
`"101"`	3	2	5	101
`B"101"`	3	2	5	101
`X"AB"`	8	16	171	10101011

4.2.8 Z's and X's

HDLs use `z` to indicate a floating value, `z` is particularly useful for describing a tristate buffer, whose output floats when the enable is 0. Recall from Section 2.6.2 that a bus can be driven by several tristate buffers, exactly one of which should be enabled. HDL Example 4.10 shows the idiom for a tristate buffer. If the buffer is enabled, the output is the same as the input. If the buffer is disabled, the output is assigned a floating value (`z`).

Similarly, HDLs use `x` to indicate an invalid logic level. If a bus is simultaneously driven to 0 and 1 by two enabled tristate buffers (or other gates), the result is `x`, indicating contention. If all the tristate buffers driving a bus are simultaneously OFF, the bus will float, indicated by `z`.

At the start of simulation, state nodes such as flip-flop outputs are initialized to an unknown state (`x` in SystemVerilog and `u` in VHDL). This is helpful to track errors caused by forgetting to reset a flip-flop before its output is used.

HDL Example 4.10 TRISTATE BUFFER

SystemVerilog

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

Notice that `y` is declared as `tri` rather than `logic`. `logic` signals can only have a single driver. Tristate busses can have multiple drivers, so they should be declared as a *net*. Two types of nets in SystemVerilog are called `tri` and `trireg`. Typically, exactly one driver on a net is active at a time, and the net takes on that value. If no driver is active, a `tri` floats (`z`), while a `trireg` retains the previous value. If no type is specified for an input or output, `tri` is assumed. Also note that a `tri` output from a module can be used as a `logic` input to another module. Section 4.7 further discusses nets with multiple drivers.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity tristate is
  port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
       en: in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of tristate is
begin
  y <= a when en else "ZZZZ";
end;
```

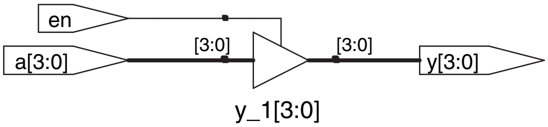


Figure 4.9 `tristate` **synthesized circuit**

If a gate receives a floating input, it may produce an `x` output when it can't determine the correct output value. Similarly, if it receives an illegal or uninitialized input, it may produce an `x` output. HDL Example 4.11

HDL Example 4.11 TRUTH TABLES WITH UNDEFINED AND FLOATING INPUTS

SystemVerilog

SystemVerilog signal values are `0`, `1`, `z`, and `x`. SystemVerilog constants starting with `z` or `x` are padded with leading `z`'s or `x`'s (instead of 0's) to reach their full length when necessary.

Table 4.5 shows a truth table for an AND gate using all four possible signal values. Note that the gate can sometimes determine the output despite some inputs being unknown. For example `0 & z` returns `0` because the output of an AND gate is always `0` if either input is `0`. Otherwise, floating or invalid inputs cause invalid outputs, displayed as `x` in SystemVerilog.

VHDL

VHDL STD_LOGIC signals are `'0'`, `'1'`, `'z'`, `'x'`, and `'u'`.

Table 4.6 shows a truth table for an AND gate using all five possible signal values. Notice that the gate can sometimes determine the output despite some inputs being unknown. For example, `'0' and 'z'` returns `'0'` because the output of an AND gate is always `'0'` if either input is `'0'`. Otherwise, floating or invalid inputs cause invalid outputs, displayed as `'x'` in VHDL. Uninitialized inputs cause uninitialized outputs, displayed as `'u'` in VHDL.

Table 4.5 SystemVerilog AND gate truth table with `z` and `x`

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

Table 4.6 VHDL AND gate truth table with `z`, `x` and `u`

AND		A				
		0	1	z	x	u
B	0	0	0	0	0	0
	1	0	1	x	x	u
	z	0	x	x	x	u
	x	0	x	x	x	u
	u	0	u	u	u	u

HDL Example 4.12 BIT SWIZZLING

SystemVerilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

The `{}` operator is used to concatenate busses. `{3{d[0]}}` indicates three copies of `d[0]`.

Don't confuse the 3-bit binary constant `3'b101` with a bus named `b`. Note that it was critical to specify the length of 3 bits in the constant; otherwise, it would have had an unknown number of leading zeros that might appear in the middle of `y`.

If `y` were wider than 9 bits, zeros would be placed in the most significant bits.

VHDL

```
y <=(c(2 downto 1), d(0), d(0), d(0), c(0), 3B"101");
```

The `()` aggregate operator is used to concatenate busses. `y` must be a 9-bit `STD_LOGIC_VECTOR`.

Another example demonstrates the power of VHDL aggregations. Assuming `z` is an 8-bit STD_LOGIC_VECTOR, `z` is given the value 10010110 using the following command aggregation.

```
z <= ("10", 4 => '1', 2 downto 1 =>'1', others =>'0')
```

The `"10"` goes in the leading pair of bits. 1s are also placed into bit 4 and bits 2 and 1. The other bits are 0.

shows how SystemVerilog and VHDL combine these different signal values in logic gates.

Seeing `x` or `u` values in simulation is almost always an indication of a bug or bad coding practice. In the synthesized circuit, this corresponds to a floating gate input, uninitialized state, or contention. The `x` or `u` may be interpreted randomly by the circuit as 0 or 1, leading to unpredictable behavior.

4.2.9 Bit Swizzling

Often it is necessary to operate on a subset of a bus or to concatenate (join together) signals to form busses. These operations are collectively known as *bit swizzling*. In HDL Example 4.12, `y` is given the 9-bit value $c_2c_1d_0d_0d_0c_0101$ using bit swizzling operations.

4.2.10 Delays

HDL statements may be associated with delays specified in arbitrary units. They are helpful during simulation to predict how fast a circuit will work (if you specify meaningful delays) and also for debugging purposes to understand cause and effect (deducing the source of a bad output is tricky if all signals change simultaneously in the simulation results). These delays are ignored during synthesis; the delay of a gate produced by the synthesizer depends on its t_{pd} and t_{cd} specifications, not on numbers in HDL code.

HDL Example 4.13 adds delays to the original function from HDL Example 4.1, $y = \overline{a}\,\overline{b}\,\overline{c} + a\overline{b}\,\overline{c} + a\overline{b}c$. It assumes that inverters have a delay of 1 ns, three-input AND gates have a delay of 2 ns, and three-input OR gates have a delay of 4 ns. Figure 4.10 shows the simulation waveforms, with `y` lagging 7 ns after the inputs. Note that `y` is initially unknown at the beginning of the simulation.

HDL Example 4.13 LOGIC GATES WITH DELAYS

SystemVerilog

```
'timescale 1ns/1ps

module example(input  logic a, b, c,
               output logic y);

  logic ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

SystemVerilog files can include a timescale directive that indicates the value of each time unit. The statement is of the form `'timescale unit/precision`. In this file, each unit is 1 ns, and the simulation has 1 ps precision. If no timescale directive is given in the file, a default unit and precision (usually 1 ns for both) are used. In SystemVerilog, a # symbol is used to indicate the number of units of delay. It can be placed in `assign` statements, as well as non-blocking (`<=`) and blocking (`=`) assignments, which will be discussed in Section 4.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity example is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of example is
  signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
  ab <= not a after 1 ns;
  bb <= not b after 1 ns;
  cb <= not c after 1 ns;
  n1 <= ab and bb and cb after 2 ns;
  n2 <= a and bb and cb after 2 ns;
  n3 <= a and bb and c after 2 ns;
  y  <= n1 or n2 or n3 after 4 ns;
end;
```

In VHDL, the `after` clause is used to indicate delay. The units, in this case, are specified as nanoseconds.

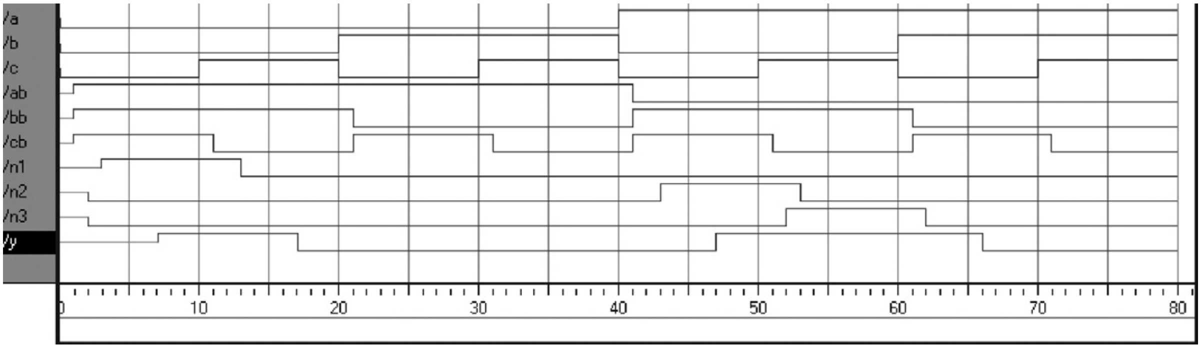


Figure 4.10 Example simulation waveforms with delays (from the ModelSim simulator)

4.3 STRUCTURAL MODELING

The previous section discussed *behavioral* modeling, describing a module in terms of the relationships between inputs and outputs. This section examines *structural* modeling, describing a module in terms of how it is composed of simpler modules.

For example, HDL Example 4.14 shows how to assemble a 4:1 multiplexer from three 2:1 multiplexers. Each copy of the 2:1 multiplexer is called

---

HDL Example 4.14 STRUCTURAL MODEL OF 4:1 MULTIPLEXER

SystemVerilog

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  logic [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

The three `mux2` instances are called `lowmux`, `highmux`, and `finalmux`. The `mux2` module must be defined elsewhere in the SystemVerilog code — see HDL Example 4.5, 4.15, or 4.34.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux4 is
  component mux2
    port(d0,
         d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
  lowmux:   mux2 port map(d0, d1, s(0), low);
  highmux:  mux2 port map(d2, d3, s(0), high);
  finalmux: mux2 port map(low, high, s(1), y);
end;
```

The architecture must first declare the `mux2` ports using the `component` declaration statement. This allows VHDL tools to check that the component you wish to use has the same ports as the entity that was declared somewhere else in another entity statement, preventing errors caused by changing the entity but not the instance. However, component declaration makes VHDL code rather cumbersome.

Note that this architecture of `mux4` was named `struct`, whereas architectures of modules with behavioral descriptions from Section 4.2 were named `synth`. VHDL allows multiple architectures (implementations) for the same entity; the architectures are distinguished by name. The names themselves have no significance to the CAD tools, but `struct` and `synth` are common. Synthesizable VHDL code generally contains only one architecture for each entity, so we will not discuss the VHDL syntax to configure which architecture is used when multiple architectures are defined.

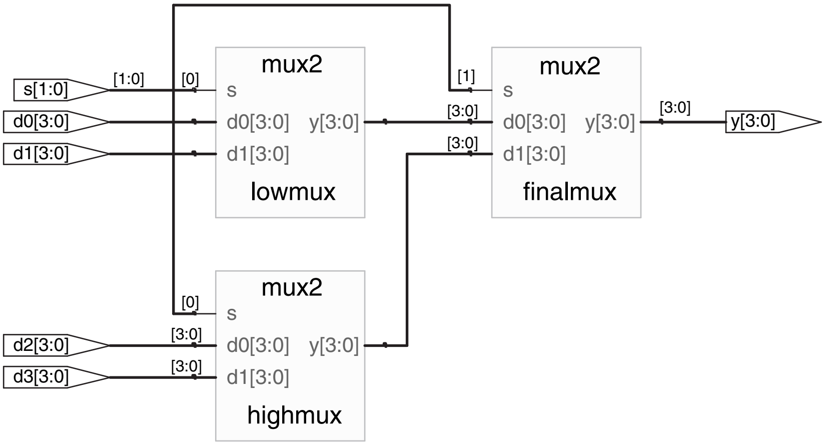


Figure 4.11 `mux4` **synthesized circuit**

an *instance*. Multiple instances of the same module are distinguished by distinct names, in this case `lowmux`, `highmux`, and `finalmux`. This is an example of regularity, in which the 2:1 multiplexer is reused many times.

HDL Example 4.15 uses structural modeling to construct a 2:1 multiplexer from a pair of tristate buffers. Building logic out of tristates is not recommended, however.

HDL Example 4.15 STRUCTURAL MODEL OF 2:1 MULTIPLEXER

SystemVerilog

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output tri   [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

In SystemVerilog, expressions such as `~s` are permitted in the port list for an instance. Arbitrarily complicated expressions are legal but discouraged because they make the code difficult to read.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux2 is
  component tristate
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal sbar: STD_LOGIC;
begin
  sbar <= not s;
  t0: tristate port map(d0, sbar, y);
  t1: tristate port map(d1, s, y);
end;
```

In VHDL, expressions such as `not s` are not permitted in the port map for an instance. Thus, `sbar` must be defined as a separate signal.

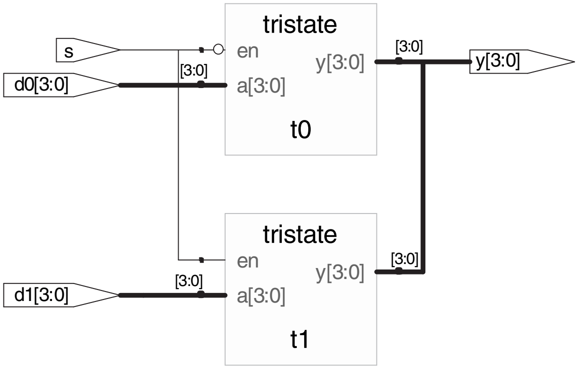


Figure 4.12 `mux2` **synthesized circuit**

HDL Example 4.16 shows how modules can access part of a bus. An 8-bit wide 2:1 multiplexer is built using two of the 4-bit 2:1 multiplexers already defined, operating on the low and high nibbles of the byte.

In general, complex systems are designed *hierarchically*. The overall system is described structurally by instantiating its major components. Each of these components is described structurally from its building blocks, and so forth recursively until the pieces are simple enough to describe behaviorally. It is good style to avoid (or at least to minimize) mixing structural and behavioral descriptions within a single module.

HDL Example 4.16 ACCESSING PARTS OF BUSSES

SystemVerilog

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2_8 is
  port(d0, d1: in  STD_LOGIC_VECTOR(7 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux2_8 is
  component mux2
    port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(3 downto 0));
  end component;
begin
  lsbmux: mux2
    port map(d0(3 downto 0), d1(3 downto 0),
             s, y(3 downto 0));
  msbmux: mux2
    port map(d0(7 downto 4), d1(7 downto 4),
             s, y(7 downto 4));
end;
```

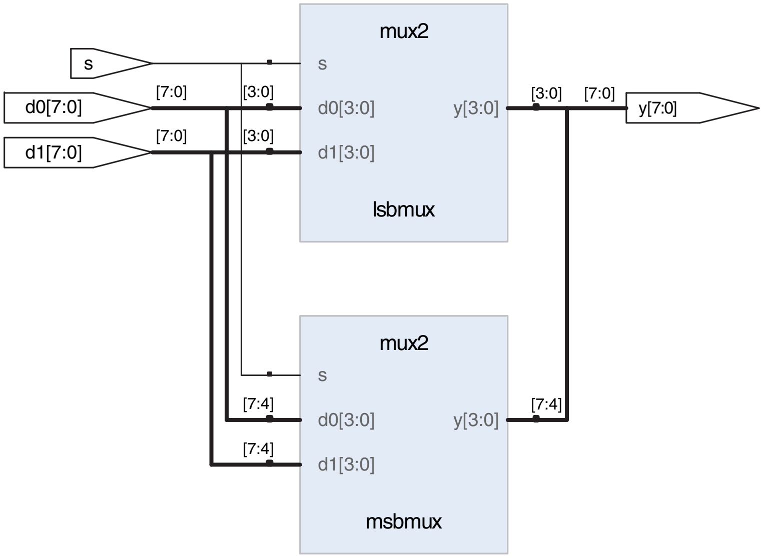


Figure 4.13 `mux2_8` **synthesized circuit**

4.4 SEQUENTIAL LOGIC

HDL synthesizers recognize certain idioms and turn them into specific sequential circuits. Other coding styles may simulate correctly but synthesize into circuits with blatant or subtle errors. This section presents the proper idioms to describe registers and latches.

4.4.1 Registers

The vast majority of modern commercial systems are built with registers using positive edge-triggered D flip-flops. HDL Example 4.17 shows the idiom for such flip-flops.

In SystemVerilog `always` statements and VHDL `process` statements, signals keep their old value until an event in the sensitivity list takes place that explicitly causes them to change. Hence, such code, with appropriate sensitivity lists, can be used to describe sequential circuits with memory. For example, the flip-flop includes only `clk` in the sensitive list. It remembers its old value of `q` until the next rising edge of the `clk`, even if `d` changes in the interim.

In contrast, SystemVerilog continuous assignment statements (`assign`) and VHDL concurrent assignment statements (<=) are reevaluated anytime any of the inputs on the right hand side changes. Therefore, such code necessarily describes combinational logic.

HDL Example 4.17 REGISTER

SystemVerilog

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;
endmodule
```

In general, a SystemVerilog `always` statement is written in the form

```
always @(sensitivity list)
  statement;
```

The `statement` is executed only when the event specified in the `sensitivity list` occurs. In this example, the statement is `q <= d` (pronounced "q gets d"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`. Note that sensitivity lists are also referred to as stimulus lists.

<= is called a *nonblocking assignment*. Think of it as a regular = sign for now; we'll return to the more subtle points in Section 4.5.4. Note that <= is used instead of `assign` inside an `always` statement.

As will be seen in subsequent sections, `always` statements can be used to imply flip-flops, latches, or combinational logic, depending on the sensitivity list and statement. Because of this flexibility, it is easy to produce the wrong hardware inadvertently. SystemVerilog introduces `always_ff`, `always_latch`, and `always_comb` to reduce the risk of common errors. `always_ff` behaves like `always` but is used exclusively to imply flip-flops and allows tools to produce a warning if anything else is implied.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flop is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of flop is
begin
  process(clk) begin
    if rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;
```

A VHDL `process` is written in the form

```
process(sensitivity list) begin
  statement;
end process;
```

The `statement` is executed when any of the variables in the `sensitivity list` change. In this example, the `if` statement checks if the change was a rising edge on clk. If so, then `q <= d` (pronounced "q gets d"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`.

An alternative VHDL idiom for a flip-flop is

```
process(clk) begin
  if clk'event and clk = '1' then
    q <= d;
  end if;
end process;
```

`rising_edge(clk)` is synonymous with `clk'event and clk = '1'`.

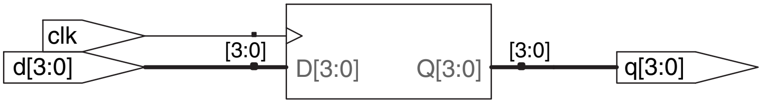


Figure 4.14 `flop` **synthesized circuit**

4.4.2 Resettable Registers

When simulation begins or power is first applied to a circuit, the output of a flop or register is unknown. This is indicated with `x` in SystemVerilog and `u` in VHDL. Generally, it is good practice to use resettable registers so that on powerup you can put your system in a known state. The reset may be either asynchronous or synchronous. Recall that asynchronous reset occurs immediately, whereas synchronous reset clears the output only on the next rising edge of the clock. HDL Example 4.18 demonstrates the idioms for flip-flops with asynchronous and synchronous resets. Note that distinguishing synchronous and asynchronous reset in a schematic can be difficult. The schematic produced by Synplify Premier places asynchronous reset at the bottom of a flip-flop and synchronous reset on the left side.

HDL Example 4.18 RESETTABLE REGISTER

SystemVerilog

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule

module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if (reset)  q <= 4'b0;
    else        q <= d;
endmodule
```

Multiple signals in an `always` statement sensitivity list are separated with a comma or the word `or`. Notice that `posedge reset` is in the sensitivity list on the asynchronously resettable flop, but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

Because the modules have the same name, `flopr`, you may include only one or the other in your design.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(3 downto 0);
       q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then
      q <= "0000";
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(3 downto 0);
       q:          out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synchronous of flopr is
begin
  process(clk) begin
    if rising_edge(clk) then
      if reset then q <= "0000";
      else q <= d;
      end if;
    end if;
  end process;
end;
```

Multiple signals in a `process` sensitivity list are separated with a comma. Notice that `reset` is in the sensitivity list on the asynchronously resettable flop, but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

Recall that the state of a flop is initialized to `'u'` at startup during VHDL simulation.

As mentioned earlier, the name of the architecture (`asynchronous` or `synchronous`, in this example) is ignored by the VHDL tools but may be helpful to the human reading the code. Because both architectures describe the entity `flopr`, you may include only one or the other in your design.

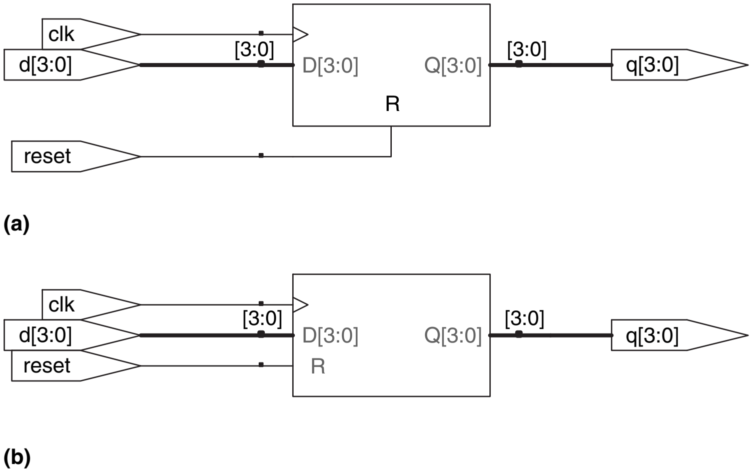


Figure 4.15 `flopr` **synthesized circuit (a) asynchronous reset, (b) synchronous reset**

4.4.3 Enabled Registers

Enabled registers respond to the clock only when the enable is asserted. HDL Example 4.19 shows an asynchronously resettable enabled register that retains its old value if both `reset` and `en` are FALSE.

HDL Example 4.19 RESETTABLE ENABLED REGISTER

SystemVerilog

```
module flopenr(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopenr is
  port(clk,
       reset,
       en: in  STD_LOGIC;
       d:  in  STD_LOGIC_VECTOR(3 downto 0);
       q:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture asynchronous of flopenr is
-- asynchronous reset
begin
  process(clk, reset) begin
    if reset then
      q <= "0000";
    elsif rising_edge(clk) then
      if en then
        q <= d;
      end if;
    end if;
  end process;
end;
```

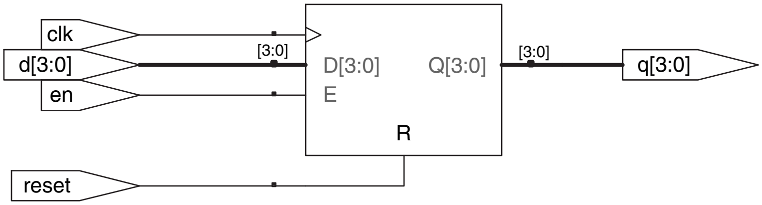


Figure 4.16 `flopenr` **synthesized circuit**

4.4.4 Multiple Registers

A single `always/process` statement can be used to describe multiple pieces of hardware. For example, consider the synchronizer from Section 3.5.5 made of two back-to-back flip-flops, as shown in Figure 4.17. HDL Example 4.20 describes the synchronizer. On the rising edge of `clk`, `d` is copied to `n1`. At the same time, `n1` is copied to `q`.

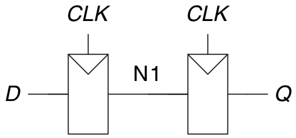


Figure 4.17 **Synchronizer circuit**

HDL Example 4.20 SYNCHRONIZER

SystemVerilog

```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 <= d; // nonblocking
      q <= n1; // nonblocking
    end
endmodule
```

Notice that the `begin/end` construct is necessary because multiple statements appear in the `always` statement. This is analogous to `{}` in C or Java. The `begin/end` was not needed in the `flopr` example because `if/else` counts as a single statement.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sync is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture good of sync is
  signal n1: STD_LOGIC;
begin
  process(clk) begin
    if rising_edge(clk) then
      n1 <= d;
      q <= n1;
    end if;
  end process;
end;
```

`n1` must be declared as a `signal` because it is an internal signal used in the module.

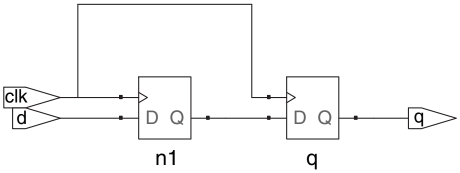


Figure 4.18 `sync` **synthesized circuit**

4.4.5 Latches

Recall from Section 3.2.2 that a D latch is transparent when the clock is HIGH, allowing data to flow from input to output. The latch becomes opaque when the clock is LOW, retaining its old state. HDL Example 4.21 shows the idiom for a D latch.

Not all synthesis tools support latches well. Unless you know that your tool does support latches and you have a good reason to use them, avoid them and use edge-triggered flip-flops instead. Furthermore, take care that your HDL does not imply any unintended latches, something that is easy to do if you aren't attentive. Many synthesis tools warn you when a latch is created; if you didn't expect one, track down the bug in your HDL. And if you don't know whether you intended to have a latch or not, you are probably approaching HDLs like a programming language and have bigger problems lurking.

4.5 MORE COMBINATIONAL LOGIC

In Section 4.2, we used assignment statements to describe combinational logic behaviorally. SystemVerilog `always` statements and VHDL `process`

HDL Example 4.21 D LATCH

SystemVerilog

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;
endmodule
```

`always_latch` is equivalent to `always @(clk, d)` and is the preferred idiom for describing a latch in SystemVerilog. It evaluates any time `clk` or `d` changes. If `clk` is HIGH, `d` flows through to `q`, so this code describes a positive level sensitive latch. Otherwise, `q` keeps its old value. SystemVerilog can generate a warning if the `always_latch` block doesn't imply a latch.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity latch is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of latch is
begin
  process(clk, d) begin
    if clk = '1' then
      q <= d;
    end if;
  end process;
end;
```

The sensitivity list contains both `clk` and `d`, so the `process` evaluates anytime `clk` or `d` changes. If `clk` is HIGH, `d` flows through to `q`.

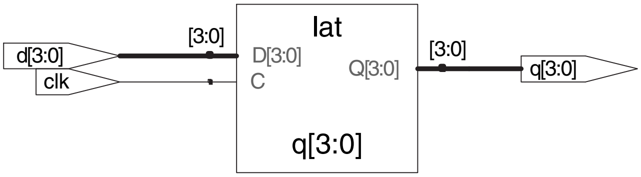


Figure 4.19 `latch` **synthesized circuit**

HDL Example 4.22 INVERTER USING `always/process`

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  always_comb
    y = ~a;
endmodule
```

`always_comb` reevaluates the statements inside the `always` statement any time any of the signals on the right hand side of <= or = in the `always` statement change. In this case, it is equivalent to `always @(a)`, but is better because it avoids mistakes if signals in the `always` statement are renamed or added. If the code inside the `always` block is not combinational logic, SystemVerilog will report a warning. `always_comb` is equivalent to `always @(*)`, but is preferred in SystemVerilog.

The = in the `always` statement is called a *blocking assignment*, in contrast to the <= nonblocking assignment. In SystemVerilog, it is good practice to use blocking assignments for combinational logic and nonblocking assignments for sequential logic. This will be discussed further in Section 4.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture proc of inv is
begin
  process(all) begin
    y <= not a;
  end process;
end;
```

`process(all)` reevaluates the statements inside the `process` any time any of the signals in the `process` change. It is equivalent to `process(a)` but is better because it avoids mistakes if signals in the `process` are renamed or added.

The `begin` and `end process` statements are required in VHDL even though the `process` contains only one assignment.

statements are used to describe sequential circuits, because they remember the old state when no new state is prescribed. However, `always/process` statements can also be used to describe combinational logic behaviorally if the sensitivity list is written to respond to changes in all of the inputs and the body prescribes the output value for every possible input combination. HDL Example 4.22 uses `always/process` statements to describe a bank of four inverters (see Figure 4.3 for the synthesized circuit).

HDLs support *blocking* and *nonblocking assignments* in an `always/process` statement. A group of blocking assignments are evaluated in the order in which they appear in the code, just as one would expect in a standard programming language. A group of nonblocking assignments are evaluated concurrently; all of the statements are evaluated before any of the signals on the left hand sides are updated.

HDL Example 4.23 defines a full adder using intermediate signals `p` and `g` to compute `s` and `cout`. It produces the same circuit from Figure 4.8, but uses `always/process` statements in place of assignment statements.

These two examples are poor applications of `always/process` statements for modeling combinational logic because they require more lines than the equivalent approach with assignment statements from HDL Examples 4.2 and 4.7. However, `case` and `if` statements are convenient for modeling more complicated combinational logic. `case` and `if` statements must appear within `always/process` statements and are examined in the next sections.

SystemVerilog

In a SystemVerilog `always` statement, = indicates a blocking assignment and <= indicates a nonblocking assignment (also called a concurrent assignment).

Do not confuse either type with continuous assignment using the `assign` statement. `assign` statements must be used outside `always` statements and are also evaluated concurrently.

VHDL

In a VHDL `process` statement, := indicates a blocking assignment and <= indicates a nonblocking assignment (also called a concurrent assignment). This is the first section where := is introduced.

Nonblocking assignments are made to outputs and to signals. Blocking assignments are made to variables, which are declared in `process` statements (see HDL Example 4.23). <= can also appear outside `process` statements, where it is also evaluated concurrently.

HDL Example 4.23 FULL ADDER USING `always/process`

SystemVerilog

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  always_comb
    begin
      p = a ^ b;              // blocking
      g = a & b;              // blocking

      s = p ^ cin;            // blocking
      cout = g | (p & cin);   // blocking
    end
endmodule
```

In this case, `always @(a, b, cin)` would have been equivalent to `always_comb`. However, `always_comb` is better because it avoids common mistakes of missing signals in the sensitivity list.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for combinational logic. This example uses blocking assignments, first computing `p`, then `g`, then `s`, and finally `cout`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
begin
  process(all)
    variable p, g: STD_LOGIC;
  begin
    p := a xor b; -- blocking
    g := a and b; -- blocking
    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

In this case, `process(a, b, cin)` would have been equivalent to `process(all)`. However, `process(all)` is better because it avoids common mistakes of missing signals in the sensitivity list.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for intermediate variables in combinational logic. This example uses blocking assignments for `p` and `g` so that they get their new values before being used to compute `s` and `cout` that depend on them.

Because `p` and `g` appear on the left hand side of a blocking assignment (:=) in a `process` statement, they must be declared to be `variable` rather than `signal`. The variable declaration appears before the `begin` in the process where the variable is used.

4.5.1 Case Statements

A better application of using the `always/process` statement for combinational logic is a seven-segment display decoder that takes advantage of the `case` statement that must appear inside an `always/process` statement.

As you might have noticed in the seven-segment display decoder of Example 2.10, the design process for large blocks of combinational logic is tedious and prone to error. HDLs offer a great improvement, allowing you to specify the function at a higher level of abstraction, and then automatically synthesize the function into gates. HDL Example 4.24 uses `case` statements to describe a seven-segment display decoder based on its truth table. The `case` statement performs different actions depending on the value of its input. A `case` statement implies combinational logic if all

---

HDL Example 4.24 SEVEN-SEGMENT DISPLAY DECODER

SystemVerilog

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case(data)
      //                  abc_defg
      0:       segments = 7'b111_1110;
      1:       segments = 7'b011_0000;
      2:       segments = 7'b110_1101;
      3:       segments = 7'b111_1001;
      4:       segments = 7'b011_0011;
      5:       segments = 7'b101_1011;
      6:       segments = 7'b101_1111;
      7:       segments = 7'b111_0000;
      8:       segments = 7'b111_1111;
      9:       segments = 7'b111_0011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```

The `case` statement checks the value of `data`. When `data` is 0, the statement performs the action after the colon, setting `segments` to 1111110. The `case` statement similarly checks other data values up to 9 (note the use of the default base, base 10).

The `default` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

In SystemVerilog, `case` statements must appear inside `always` statements.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity seven_seg_decoder is
  port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_seg_decoder is
begin
  process(all) begin
    case data is
      --                         abcdefg
      when X"0" =>    segments <= "1111110";
      when X"1" =>    segments <= "0110000";
      when X"2" =>    segments <= "1101101";
      when X"3" =>    segments <= "1111001";
      when X"4" =>    segments <= "0110011";
      when X"5" =>    segments <= "1011011";
      when X"6" =>    segments <= "1011111";
      when X"7" =>    segments <= "1110000";
      when X"8" =>    segments <= "1111111";
      when X"9" =>    segments <= "1110011";
      when others =>  segments <= "0000000";
    end case;
  end process;
end;
```

The `case` statement checks the value of `data`. When `data` is 0, the statement performs the action after the =>, setting `segments` to 1111110. The `case` statement similarly checks other data values up to 9 (note the use of `X` for hexadecimal numbers). The `others` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

Unlike SystemVerilog, VHDL supports selected signal assignment statements (see HDL Example 4.6), which are much like `case` statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic.

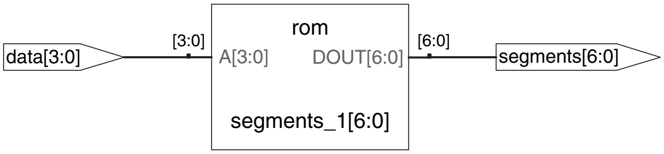


Figure 4.20 `sevenseg` **synthesized circuit**

possible input combinations are defined; otherwise it implies sequential logic, because the output will keep its old value in the undefined cases.

Synplify Premier synthesizes the seven-segment display decoder into a *read-only memory* (*ROM*) containing the 7 outputs for each of the 16 possible inputs. ROMs are discussed further in Section 5.5.6.

If the `default` or `others` clause were left out of the `case` statement, the decoder would have remembered its previous output anytime data were in the range of 10–15. This is strange behavior for hardware.

Ordinary decoders are also commonly written with `case` statements. HDL Example 4.25 describes a 3:8 decoder.

4.5.2 If Statements

`always/process` statements may also contain `if` statements. The `if` statement may be followed by an `else` statement. If all possible input combinations

HDL Example 4.25 3:8 DECODER

SystemVerilog

```
module decoder3_8(input  logic [2:0] a,
                  output logic [7:0] y);
  always_comb
    case(a)
      3'b000:  y = 8'b00000001;
      3'b001:  y = 8'b00000010;
      3'b010:  y = 8'b00000100;
      3'b011:  y = 8'b00001000;
      3'b100:  y = 8'b00010000;
      3'b101:  y = 8'b00100000;
      3'b110:  y = 8'b01000000;
      3'b111:  y = 8'b10000000;
      default: y = 8'bxxxxxxxx;
    endcase
endmodule
```

The `default` statement isn't strictly necessary for logic synthesis in this case because all possible input combinations are defined, but it is prudent for simulation in case one of the inputs is an `x` or `z`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity decoder3_8 is
  port(a: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of decoder3_8 is
begin
  process(all) begin
    case a is
      when "000" =>   y <= "00000001";
      when "001" =>   y <= "00000010";
      when "010" =>   y <= "00000100";
      when "011" =>   y <= "00001000";
      when "100" =>   y <= "00010000";
      when "101" =>   y <= "00100000";
      when "110" =>   y <= "01000000";
      when "111" =>   y <= "10000000";
      when others =>  y <= "XXXXXXXX";
    end case;
  end process;
end;
```

The `others` clause isn't strictly necessary for logic synthesis in this case because all possible input combinations are defined, but it is prudent for simulation in case one of the inputs is an `x`, `z`, or `u`.

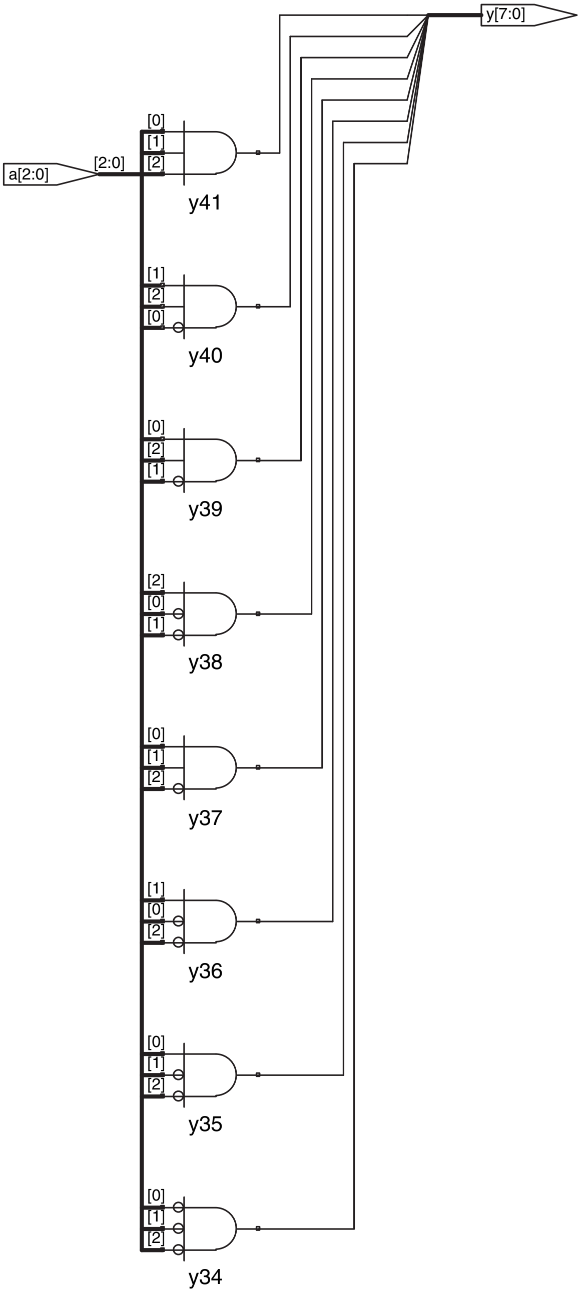

y[7:0]
a[2:0]
[2:0]
[0]
[1]
[2]
y41
[1]
[2]
[0]
y40
[0]
[2]
[1]
y39
[2]
[0]
[1]
y38
[0]
[1]
[2]
y37
[1]
[0]
[2]
y36
[0]
[1]
[2]
y35
[0]
[1]
[2]
y34

Figure 4.21 `decoder3_8` **synthesized circuit**

HDL Example 4.26 PRIORITY CIRCUIT

SystemVerilog

```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);

  always_comb
    if      (a[3]) y <= 4'b1000;
    else if (a[2]) y <= 4'b0100;
    else if (a[1]) y <= 4'b0010;
    else if (a[0]) y <= 4'b0001;
    else           y <= 4'b0000;
endmodule
```

In SystemVerilog, `if` statements must appear inside of `always` statements.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priorityckt is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of priorityckt is
begin
  process(all) begin
    if    a(3) then y <= "1000";
    elsif a(2) then y <= "0100";
    elsif a(1) then y <= "0010";
    elsif a(0) then y <= "0001";
    else            y <= "0000";
    end if;
  end process;
end;
```

Unlike SystemVerilog, VHDL supports conditional signal assignment statements (see HDL Example 4.6), which are much like `if` statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic.

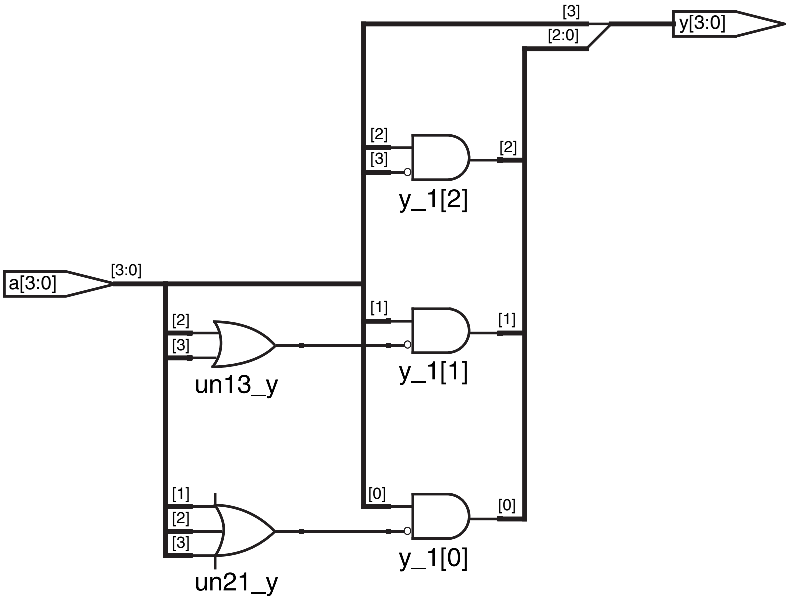


Figure 4.22 `priorityckt` **synthesized circuit**

are handled, the statement implies combinational logic; otherwise, it produces sequential logic (like the latch in Section 4.4.5).

HDL Example 4.26 uses `if` statements to describe a priority circuit, defined in Section 2.4. Recall that an *N*-input priority circuit sets the output TRUE that corresponds to the most significant input that is TRUE.

4.5.3 Truth Tables with Don't Cares

As examined in Section 2.7.3, truth tables may include don't care's to allow more logic simplification. HDL Example 4.27 shows how to describe a priority circuit with don't cares.

Synplify Premier synthesizes a slightly different circuit for this module, shown in Figure 4.23, than it did for the priority circuit in Figure 4.22. However, the circuits are logically equivalent.

HDL Example 4.27 PRIORITY CIRCUIT USING DON'T CARES

SystemVerilog

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y <= 4'b1000;
      4'b01??: y <= 4'b0100;
      4'b001?: y <= 4'b0010;
      4'b0001: y <= 4'b0001;
      default: y <= 4'b0000;
    endcase
endmodule
```

The `casez` statement acts like a `case` statement except that it also recognizes `?` as don't care.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priority_casez is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture dontcare of priority_casez is
begin
  process(all) begin
    case? a is
      when "1---" => y <= "1000";
      when "01--" => y <= "0100";
      when "001-" => y <= "0010";
      when "0001" => y <= "0001";
      when others => y <= "0000";
    end case?;
  end process;
end;
```

The `case?` statement acts like a `case` statement except that it also recognizes `-` as don't care.

4.5.4 Blocking and Nonblocking Assignments

The guidelines on page 206 explain when and how to use each type of assignment. If these guidelines are not followed, it is possible to write code that appears to work in simulation but synthesizes to incorrect hardware. The optional remainder of this section explains the principles behind the guidelines.

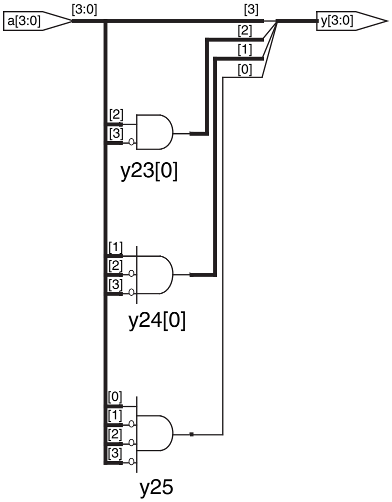


Figure 4.23 `priority_casez` **synthesized circuit**

BLOCKING AND NONBLOCKING ASSIGNMENT GUIDELINES

SystemVerilog

1. Use `always_ff @(posedge clk)` and nonblocking assignments to model synchronous sequential logic.

```
always_ff @(posedge clk)
  begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
  end
```

2. Use continuous assignments to model simple combinational logic.

```
assign y = s ? d1 : d0;
```

3. Use `always_comb` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful.

```
always_comb
  begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
  end
```

4. Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.

VHDL

1. Use `process(clk)` and nonblocking assignments to model synchronous sequential logic.

```
process(clk) begin
  if rising_edge(clk) then
    n1 <= d; -- nonblocking
    q <= n1; -- nonblocking
  end if;
end process;
```

2. Use concurrent assignments outside `process` statements to model simple combinational logic.

```
y <= d0 when s = '0' else d1;
```

3. Use `process(all)` to model more complicated combinational logic where the `process` is helpful. Use blocking assignments for internal variables.

```
process(all)
  variable p, g: STD_LOGIC;
begin
  p := a xor b; -- blocking
  g := a and b; -- blocking
  s <= p xor cin;
  cout <= g or (p and cin);
end process;
```

4. Do not make assignments to the same variable in more than one `process` or concurrent assignment statement.

Combinational Logic*

The full adder from HDL Example 4.23 is correctly modeled using blocking assignments. This section explores how it operates and how it would differ if nonblocking assignments had been used.

Imagine that `a`, `b`, and `cin` are all initially 0. `p`, `g`, `s`, and `cout` are thus 0 as well. At some time, `a` changes to 1, triggering the `always/process` statement. The four blocking assignments evaluate in the order shown here. (In the VHDL code, `s` and `cout` are assigned concurrently.) Note that `p` and `g` get their new values before `s` and `cout` are computed because of the blocking assignments. This is important because we want to compute `s` and `cout` using the new values of `p` and `g`.

1. $\texttt{p} \leftarrow 1 \oplus 0 = 1$
2. $\texttt{g} \leftarrow 1 \bullet 0 = 0$
3. $\texttt{s} \leftarrow 1 \oplus 0 = 1$
4. $\texttt{cout} \leftarrow 0 + 1 \bullet 0 = 0$

In contrast, HDL Example 4.28 illustrates the use of nonblocking assignments.

Now consider the same case of `a` rising from 0 to 1 while `b` and `cin` are 0. The four nonblocking assignments evaluate concurrently:

$$\texttt{p} \leftarrow 1 \oplus 0 = 1 \quad \texttt{g} \leftarrow 1 \bullet 0 = 0 \quad \texttt{s} \leftarrow 0 \oplus 0 = 0 \quad \texttt{cout} \leftarrow 0 + 0 \bullet 0 = 0$$

HDL Example 4.28 FULL ADDER USING NONBLOCKING ASSIGNMENTS

SystemVerilog

```
// nonblocking assignments (not recommended)
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  always_comb
    begin
      p <= a ^ b; // nonblocking
      g <= a & b; // nonblocking

      s <= p ^ cin;
      cout <= g | (p & cin);
    end
endmodule
```

VHDL

```
-- nonblocking assignments (not recommended)
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture nonblocking of fulladder is
  signal p, g: STD_LOGIC;
begin
  process(all) begin
    p <= a xor b; -- nonblocking
    g <= a and b; -- nonblocking
    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

Because `p` and `g` appear on the left hand side of a nonblocking assignment in a `process` statement, they must be declared to be `signal` rather than `variable`. The `signal` declaration appears before the `begin` in the `architecture`, not the `process`.

Observe that `s` is computed concurrently with `p` and hence uses the old value of `p`, not the new value. Therefore, `s` remains 0 rather than becoming 1. However, `p` does change from 0 to 1. This change triggers the `always/process` statement to evaluate a second time, as follows:

$$\texttt{p} \leftarrow 1 \oplus 0 = 1 \quad \texttt{g} \leftarrow 1 \bullet 0 = 0 \quad \texttt{s} \leftarrow 1 \oplus 0 = 1 \quad \texttt{cout} \leftarrow 0 + 1 \bullet 0 = 0$$

This time, `p` is already 1, so `s` correctly changes to 1. The nonblocking assignments eventually reach the right answer, but the `always/process` statement had to evaluate twice. This makes simulation slower, though it synthesizes to the same hardware.

Another drawback of nonblocking assignments in modeling combinational logic is that the HDL will produce the wrong result if you forget to include the intermediate variables in the sensitivity list.

Worse yet, some synthesis tools will synthesize the correct hardware even when a faulty sensitivity list causes incorrect simulation. This leads to a mismatch between the simulation results and what the hardware actually does.

SystemVerilog

If the sensitivity list of the `always` statement in HDL Example 4.28 were written as `always @(a, b, cin)` rather than `always_comb`, then the statement would not reevaluate when `p` or `g` changes. In that case, `s` would be incorrectly left at 0, not 1.

VHDL

If the sensitivity list of the `process` statement in HDL Example 4.28 were written as `process(a, b, cin)` rather than `process(all)`, then the statement would not reevaluate when `p` or `g` changes. In that case, `s` would be incorrectly left at 0, not 1.

Sequential Logic*

The synchronizer from HDL Example 4.20 is correctly modeled using nonblocking assignments. On the rising edge of the clock, `d` is copied to `n1` at the same time that `n1` is copied to `q`, so the code properly describes two registers. For example, suppose initially that `d` = 0, `n1` = 1, and `q` = 0. On the rising edge of the clock, the following two assignments occur concurrently, so that after the clock edge, `n1` = 0 and `q` = 1.

$$\texttt{n1} \leftarrow \texttt{d} = 0 \quad \texttt{q} \leftarrow \texttt{n1} = 1$$

HDL Example 4.29 tries to describe the same module using blocking assignments. On the rising edge of `clk`, `d` is copied to `n1`. Then this new value of `n1` is copied to `q`, resulting in `d` improperly appearing at both `n1` and `q`. The assignments occur one after the other so that after the clock edge, `q` = `n1` = 0.

1. $\texttt{n1} \leftarrow \texttt{d} = 0$
2. $\texttt{q} \leftarrow \texttt{n1} = 0$

HDL Example 4.29 BAD SYNCHRONIZER WITH BLOCKING ASSIGNMENTS

SystemVerilog

```
// Bad implementation of a synchronizer using blocking
// assignments

module syncbad(input  logic clk,
               input  logic d,
               output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 = d; // blocking
      q = n1; // blocking
    end
endmodule
```

VHDL

```
-- Bad implementation of a synchronizer using blocking
-- assignment

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity syncbad is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture bad of syncbad is
begin
  process(clk)
    variable n1: STD_LOGIC;
  begin
    if rising_edge(clk) then
      n1 := d; -- blocking
      q <= n1;
    end if;
  end process;
end;
```

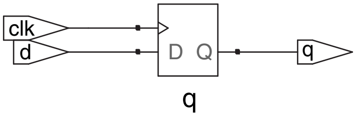


Figure 4.24 `syncbad` **synthesized circuit**

Because `n1` is invisible to the outside world and does not influence the behavior of `q`, the synthesizer optimizes it away entirely, as shown in Figure 4.24.

The moral of this illustration is to exclusively use nonblocking assignment in `always/process` statements when modeling sequential logic. With sufficient cleverness, such as reversing the orders of the assignments, you could make blocking assignments work correctly, but blocking assignments offer no advantages and only introduce the risk of unintended behavior. Certain sequential circuits will not work with blocking assignments no matter what the order.

4.6 FINITE STATE MACHINES

Recall that a finite state machine (FSM) consists of a state register and two blocks of combinational logic to compute the next state and the output given the current state and the input, as was shown in Figure 3.22. HDL descriptions of state machines are correspondingly divided into three parts to model the state register, the next state logic, and the output logic.

HDL Example 4.30 DIVIDE-BY-3 FINITE STATE MACHINE

SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype [1:0] state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0:      nextstate <= S1;
      S1:      nextstate <= S2;
      S2:      nextstate <= S0;
      default: nextstate <= S0;
    endcase

  // output logic
  assign y = (state == S0);
endmodule
```

The `typedef` statement defines `statetype` to be a two-bit `logic` value with three possibilities: `S0`, `S1`, or `S2`. `state` and `nextstate` are `statetype` signals.

The enumerated encodings default to numerical order: `S0` = 00, `S1` = 01, and `S2` = 10. The encodings can be explicitly set by the user; however, the synthesis tool views them as suggestions, not requirements. For example, the following snippet encodes the states as 3-bit one-hot values:

```
typedef enum logic [2:0] {S0 = 3'b001, S1 = 3'b010, S2 = 3'b100}
statetype;
```

Notice how a `case` statement is used to define the state transition table. Because the next state logic should be combinational, a `default` is necessary even though the state `2'b11` should never arise.

The output, `y`, is 1 when the state is `S0`. The *equality comparison* `a == b` evaluates to 1 if `a` equals `b` and 0 otherwise. The *inequality comparison* `a != b` does the inverse, evaluating to 1 if `a` does not equal `b`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity divideby3FSM is
  port(clk, reset: in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  nextstate <= S1 when state = S0 else
               S2 when state = S1 else
               S0;

  -- output logic
  y <= '1' when state = S0 else '0';
end;
```

This example defines a new *enumeration* data type, `statetype`, with three possibilities: `S0`, `S1`, and `S2`. `state` and `nextstate` are `statetype` signals. By using an enumeration instead of choosing the state encoding, VHDL frees the synthesizer to explore various state encodings to choose the best one.

The output, `y`, is 1 when the `state` is `S0`. The inequality comparison uses /=. To produce an output of 1 when the state is anything but `S0`, change the comparison to `state /= S0`.

HDL Example 4.30 describes the divide-by-3 FSM from Section 3.4.2. It provides an asynchronous reset to initialize the FSM. The state register uses the ordinary idiom for flip-flops. The next state and output logic blocks are combinational.

The Synplify Premier synthesis tool just produces a block diagram and state transition diagram for state machines; it does not show the logic

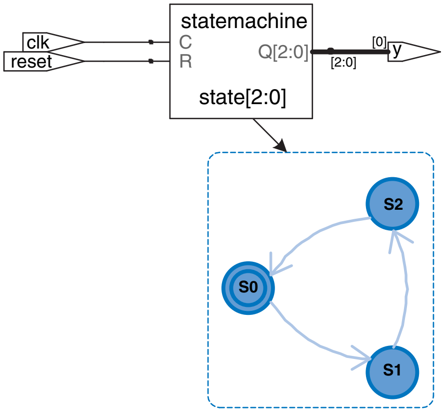


Figure 4.25 `divideby3fsm` **synthesized circuit**

gates or the inputs and outputs on the arcs and states. Therefore, be careful that you have specified the FSM correctly in your HDL code. The state transition diagram in Figure 4.25 for the divide-by-3 FSM is analogous to the diagram in Figure 3.28(b). The double circle indicates that S0 is the reset state. Gate-level implementations of the divide-by-3 FSM were shown in Section 3.4.2.

Notice that the states are named with an enumeration data type rather than by referring to them as binary values. This makes the code more readable and easier to change.

If, for some reason, we had wanted the output to be HIGH in states S0 and S1, the output logic would be modified as follows.

Notice that the synthesis tool uses a 3-bit encoding (`Q[2:0]`) instead of the 2-bit encoding suggested in the SystemVerilog code.

SystemVerilog

```
// output logic
assign y = (state == S0 | state == S1);
```

VHDL

```
-- output logic
y <= '1' when (state = S0 or state = S1) else '0';
```

The next two examples describe the snail pattern recognizer FSM from Section 3.4.3. The code shows how to use `case` and `if` statements to handle next state and output logic that depend on the inputs as well as the current state. We show both Moore and Mealy modules. In the Moore machine (HDL Example 4.31), the output depends only on the current state, whereas in the Mealy machine (HDL Example 4.32), the output logic depends on both the current state and inputs.

HDL Example 4.31 PATTERN RECOGNIZER MOORE FSM

SystemVerilog

```
module patternMoore(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S2;
          else   nextstate = S1;
      S2: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (state == S2);
endmodule
```

Note how nonblocking assignments (`<=`) are used in the state register to describe sequential logic, whereas blocking assignments (`=`) are used in the next state logic to describe combinational logic.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMoore is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMoore is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then                 state <= S0;
    elsif rising_edge(clk) then state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when S0 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when S1 =>
        if a then nextstate <= S2;
        else      nextstate <= S1;
        end if;
      when S2 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when others =>
                  nextstate <= S0;
    end case;
  end process;

  --output logic
  y <= '1' when state = S2 else '0';
end;
```

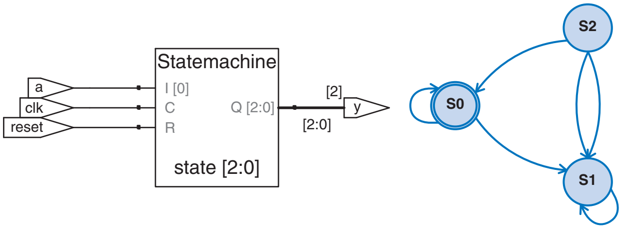


Figure 4.26 `patternMoore` **synthesized circuit**

HDL Example 4.32 PATTERN RECOGNIZER MEALY FSM

SystemVerilog

```
module patternMealy(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic {S0, S1} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (a & state == S1);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMealy is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMealy is
  type statetype is (S0, S1);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset then                  state <= S0;
    elsif rising_edge(clk) then state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(all) begin
    case state is
      when S0 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when S1 =>
        if a then nextstate <= S0;
        else      nextstate <= S1;
        end if;
      when others =>
                  nextstate <= S0;
    end case;
  end process;

  -- output logic
  y <= '1' when (a = '1' and state = S1) else '0';
end;
```

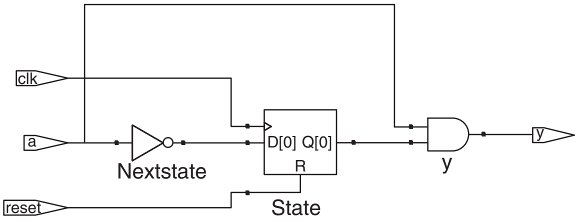


Figure 4.27 `patternMealy` **synthesized circuit**

4.7 DATA TYPES*

This section explains some subtleties about SystemVerilog and VHDL types in more depth.

4.7.1 SystemVerilog

Prior to SystemVerilog, Verilog primarily used two types: `reg` and `wire`. Despite its name, a `reg` signal might or might not be associated with a register. This was a great source of confusion for those learning the language. SystemVerilog introduced the `logic` type to eliminate the confusion; hence, this book emphasizes the `logic` type. This section explains the `reg` and `wire` types in more detail for those who need to read old Verilog code.

In Verilog, if a signal appears on the left hand side of <= or = in an `always` block, it must be declared as `reg`. Otherwise, it should be declared as `wire`. Hence, a `reg` signal might be the output of a flip-flop, a latch, or combinational logic, depending on the sensitivity list and statement of an `always` block.

Input and output ports default to the `wire` type unless their type is explicitly defined as `reg`. The following example shows how a flip-flop is described in conventional Verilog. Note that `clk` and `d` default to `wire`, while `q` is explicitly defined as `reg` because it appears on the left hand side of <= in the `always` block.

```
module flop(input            clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @(posedge clk)
    q <= d;
endmodule
```

SystemVerilog introduces the `logic` type. `logic` is a synonym for `reg` and avoids misleading users about whether it is actually a flip-flop. Moreover, SystemVerilog relaxes the rules on `assign` statements and hierarchical port instantiations so `logic` can be used outside `always` blocks where a `wire` traditionally would have been required. Thus, nearly all SystemVerilog signals can be `logic`. The exception is that signals with multiple drivers (e.g., a tristate bus) must be declared as a net, as described in HDL Example 4.10. This rule allows SystemVerilog to generate an error message rather than an `x` value when a `logic` signal is accidentally connected to multiple drivers.

The most common type of net is called a `wire` or `tri`. These two types are synonymous, but `wire` is conventionally used when a single driver is present and `tri` is used when multiple drivers are present. Thus, `wire` is obsolete in SystemVerilog because `logic` is preferred for signals with a single driver.

When a `tri` net is driven to a single value by one or more drivers, it takes on that value. When it is undriven, it floats (`z`). When it is driven to a different value (`0`, `1`, or `x`) by multiple drivers, it is in contention (`x`).

There are other net types that resolve differently when undriven or driven by multiple sources. These other types are rarely used, but may be substituted anywhere a `tri` net would normally appear (e.g., for signals with multiple drivers). Each is described in Table 4.7.

Table 4.7 **Net Resolution**

Net Type	No Driver	Conflicting Drivers
`tri`	`z`	`x`
`trireg`	`previous value`	`x`
`triand`	`z`	`0 if there are any 0`
`trior`	`z`	`1 if there are any 1`
`tri0`	`0`	`x`
`tri1`	`1`	`x`

4.7.2 VHDL

Unlike SystemVerilog, VHDL enforces a strict data typing system that can protect the user from some errors but that is also clumsy at times.

Despite its fundamental importance, the `STD_LOGIC` type is not built into VHDL. Instead, it is part of the `IEEE.STD_LOGIC_1164` library. Thus, every file must contain the library statements shown in the previous examples.

Moreover, `IEEE.STD_LOGIC_1164` lacks basic operations such as addition, comparison, shifts, and conversion to integers for the `STD_ LOGIC_VECTOR` data. These were finally added to the VHDL 2008 standard in the `IEEE.NUMERIC_STD_UNSIGNED` library.

VHDL also has a `BOOLEAN` type with two values: `true` and `false`. `BOOLEAN` values are returned by comparisons (such as the equality comparison, `s = '0'`) and are used in conditional statements such as `when` and `if`. Despite the temptation to believe a `BOOLEAN true` value should be equivalent to a `STD_LOGIC '1'` and `BOOLEAN false` should mean `STD_LOGIC '0'`, these types were not interchangeable prior to VHDL 2008. For example, in old VHDL code, one must write

```
y <= d1 when (s = '1') else d0;
```

while in VHDL 2008, the `when` statement automatically converts `s` from `STD_LOGIC` to `BOOLEAN` so one can simply write

```
y <= d1 when s else d0;
```

Even in VHDL 2008, it is still necessary to write

```
q <= '1' when (state = S2) else '0';
```

instead of

```
q <= (state = S2);
```

because `(state = S2)` returns a `BOOLEAN` result, which cannot be directly assigned to the `STD_LOGIC` signal `y`.

Although we do not declare any signals to be `BOOLEAN`, they are automatically implied by comparisons and used by conditional statements. Similarly, VHDL has an `INTEGER` type that represents both positive and negative integers. Signals of type `INTEGER` span at least the values $-(2^{31} - 1)$ to $2^{31} - 1$. Integer values are used as indices of busses. For example, in the statement

```
y <= a(3) and a(2) and a(1) and a(0);
```

0, 1, 2, and 3 are integers serving as an index to choose bits of the `a` signal. We cannot directly index a bus with a `STD_LOGIC` or `STD_ LOGIC_ VECTOR` signal. Instead, we must convert the signal to an `INTEGER`. This is demonstrated in the example below for an 8:1 multiplexer that selects one bit from a vector using a 3-bit index. The `TO_INTEGER` function is defined in the `IEEE.NUMERIC_STD_UNSIGNED` library and performs the conversion from `STD_LOGIC_VECTOR` to `INTEGER` for positive (unsigned) values.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity mux8 is
  port(d: in  STD_LOGIC_VECTOR(7 downto 0);
       s: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of mux8 is
begin
  y <= d(TO_INTEGER(s));
end;
```

VHDL is also strict about `out` ports being exclusively for output. For example, the following code for two- and three-input AND gates is illegal VHDL because `v` is an output and is also used to compute `w`.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and23 is
  port(a, b, c: in STD_LOGIC;
       v, w: out  STD_LOGIC);
end;

architecture synth of and23 is
begin
  v <= a and b;
  w <= v and c;
end;
```

VHDL defines a special port type, `buffer`, to solve this problem. A signal connected to a `buffer` port behaves as an output but may also be used within the module. The corrected entity definition follows. Verilog and SystemVerilog do not have this limitation and do not require buffer ports. VHDL 2008 eliminates this restriction by allowing `out` ports to be readable, but this change is not supported by the Synplify CAD tool at the time of this writing.

```
entity and23 is
  port(a, b, c: in STD_LOGIC;
       v: buffer  STD_LOGIC;
       w: out     STD_LOGIC);
end;
```

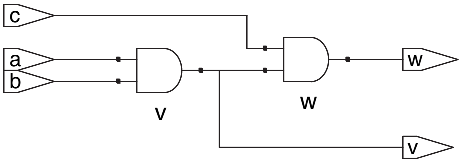


Figure 4.28 `and23` **synthesized circuit**

Most operations such as addition, subtraction, and Boolean logic are identical whether a number is signed or unsigned. However, magnitude comparison, multiplication, and arithmetic right shifts are performed differently for signed two's complement numbers than for unsigned binary numbers. These operations will be examined in Chapter 5. HDL Example 4.33 describes how to indicate that a signal represents a signed number.

HDL Example 4.33 (a) UNSIGNED MULTIPLIER (b) SIGNED MULTIPLIER

SystemVerilog

```
// 4.33(a): unsigned multiplier
module multiplier(input  logic [3:0] a, b,
                  output logic [7:0] y);
  assign y = a * b;
endmodule

// 4.33(b): signed multiplier
module multiplier(input  logic signed [3:0] a, b,
                  output logic signed [7:0] y);
  assign y = a * b;
endmodule
```

In SystemVerilog, signals are considered unsigned by default. Adding the `signed` modifier (e.g., `logic signed [3:0] a`) causes the signal `a` to be treated as signed.

VHDL

```
-- 4.33(a): unsigned multiplier
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity multiplier is
  port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
       y:    out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of multiplier is
begin
  y <= a * b;
end;
```

VHDL uses the `NUMERIC_STD_UNSIGNED` library to perform arithmetic and comparison operations on `STD_LOGIC_VECTOR`s. The vectors are treated as unsigned.

```
use IEEE.NUMERIC_STD_UNSIGNED.all;
```

VHDL also defines `UNSIGNED` and `SIGNED` data types in the `IEEE.NUMERIC_STD` library, but these involve type conversions beyond the scope of this chapter.

4.8 PARAMETERIZED MODULES*

So far all of our modules have had fixed-width inputs and outputs. For example, we had to define separate modules for 4- and 8-bit wide 2:1 multiplexers. HDLs permit variable bit widths using parameterized modules.

HDL Example 4.34 declares a parameterized 2:1 multiplexer with a default width of 8, then uses it to create 8- and 12-bit 4:1 multiplexers.

HDL Example 4.34 PARAMETERIZED *N*-BIT 2:1 MULTIPLEXERS

SystemVerilog

```
module mux2
  #(parameter width = 8)
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);

   assign y = s ? d1 : d0;
endmodule
```

SystemVerilog allows a `#(parameter . . . )` statement before the inputs and outputs to define parameters. The `parameter` statement includes a default value (8) of the parameter, in this case called `width`. The number of bits in the inputs and outputs can depend on this parameter.

```
module mux4_8(input  logic [7:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [7:0] y);

  logic [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[0], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

The 8-bit 4:1 multiplexer instantiates three 2:1 multiplexers using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `#( )` before the instance name, as shown below.

```
module mux4_12(input logic  [11:0] d0, d1, d2, d3,
               input logic  [1:0]  s,
               output logic [11:0] y);

  logic [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

Do not confuse the use of the # sign indicating delays with the use of `#(. . .)` in defining and overriding parameters.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  generic(width: integer := 8);
  port(d0,
    d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
    s:  in  STD_LOGIC;
    y:  out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

The `generic` statement includes a default value (8) of `width`. The value is an integer.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4_8 is
  port(d0, d1, d2,
       d3: in  STD_LOGIC_VECTOR(7 downto 0);
        s: in  STD_LOGIC_VECTOR(1 downto 0);
        y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux4_8 is
  component mux2
    generic(width: integer := 8);
    port(d0,
         d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal low, hi: STD_LOGIC_VECTOR(7 downto 0);
begin
  lowmux: mux2 port map(d0, d1, s(0), low);
  himux:  mux2 port map(d2, d3, s(0), hi);
  outmux: mux2 port map(low, hi, s(1), y);
end;
```

The 8-bit 4:1 multiplexer, `mux4_8`, instantiates three 2:1 multiplexers using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `generic map`, as shown below.

```
lowmux: mux2 generic map(12)
             port map(d0, d1, s(0), low);
himux:  mux2 generic map(12)
             port map(d2, d3, s(0), hi);
outmux: mux2 generic map(12)
             port map(low, hi, s(1), y);
```

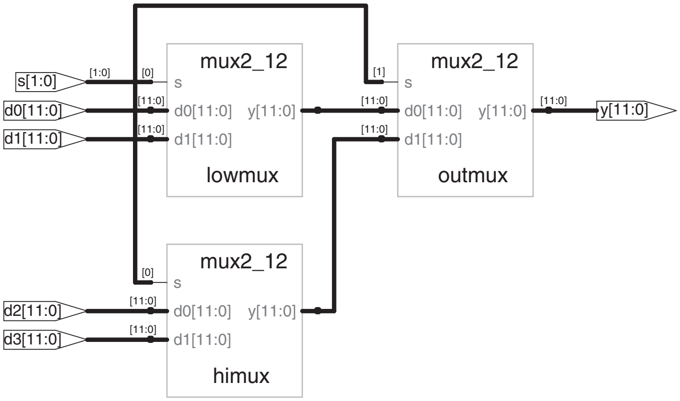

s[1:0]
d0[11:0]
d1[11:0]
d2[11:0]
d3[11:0]
[1:0]
[0]
[11:0]
[11:0]
mux2_12
s
d0[11:0]
d1[11:0]
y[11:0]
lowmux
[1]
[11:0]
[11:0]
mux2_12
s
d0[11:0]
d1[11:0]
y[11:0]
outmux
[11:0]
y[11:0]
[0]
[11:0]
[11:0]
mux2_12
s
d0[11:0]
d1[11:0]
y[11:0]
himux

Figure 4.29 `mux4_12` **synthesized circuit**

HDL Example 4.35 PARAMETERIZED *N*:2^N DECODER

SystemVerilog

```
module decoder
  #(parameter N = 3)
   (input  logic [N-1:0]    a,
    output logic [2**N-1:0] y);

  always_comb
    begin
      y = 0;
      y[a] = 1;
    end
endmodule
```

`2**N` indicates 2^N.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE. NUMERIC_STD_UNSIGNED.all;

entity decoder is
  generic(N: integer := 3);
  port(a: in  STD_LOGIC_VECTOR(N-1 downto 0);
        y: out STD_LOGIC_VECTOR(2**N-1 downto 0));
end;

architecture synth of decoder is
begin
  process(all)
  begin
    y <= (OTHERS => '0');
    y(TO_INTEGER(a)) <= '1';
  end process;
end;
```

`2**N` indicates 2^N.

HDL Example 4.35 shows a decoder, which is an even better application of parameterized modules. A large *N*:2^N decoder is cumbersome to specify with `case` statements, but easy using parameterized code that simply sets the appropriate output bit to 1. Specifically, the decoder uses blocking assignments to set all the bits to 0, then changes the appropriate bit to 1.

HDLs also provide `generate` statements to produce a variable amount of hardware depending on the value of a parameter. `generate` supports `for` loops and `if` statements to determine how many of what types of hardware to produce. HDL Example 4.36 demonstrates how to use `generate` statements to produce an *N*-input AND function from a cascade of two-input AND gates. Of course, a reduction operator would be cleaner and simpler for this application, but the example illustrates the general principle of hardware generators.

Use `generate` statements with caution; it is easy to produce a large amount of hardware unintentionally!

HDL Example 4.36 PARAMETERIZED *N*-INPUT AND GATE

SystemVerilog

```
module andN
  #(parameter width=8)
   (input  logic [width-1:0] a,
    output logic             y);

  genvar i;
  logic [width-1:0] x;

  generate
    assign x[0]=a[0];
    for(i=1; i<width; i=i+1) begin: forloop
      assign x[i]=a[i] & x[i-1];
    end
  endgenerate

  assign y=x[width-1];
endmodule
```

The `for` statement loops thrugh `i` = 1, 2, … , `width-1` to produce many consecutive AND gates. The `begin` in a `generate` `for` loop must be followed by a `:` and an arbitrary label (`forloop`, in this case).

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity andN is
  generic(width: integer := 8);
  port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of andN is
  signal x: STD_LOGIC_VECTOR(width-1 downto 0);
begin
  x(0) <= a(0);
  gen: for i in 1 to width-1 generate
    x(i) <= a(i) and x(i-1);
  end generate;
  y <= x(width-1);
end;
```

The generate loop variable `i` does not need to be declared.

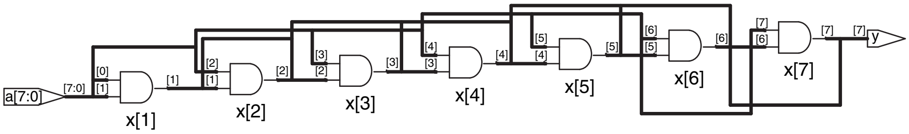


Figure 4.30 `andN` **synthesized circuit**

Some tools also call the module to be tested the *unit under test* (*UUT*).

4.9 TESTBENCHES

A *testbench* is an HDL module that is used to test another module, called the *device under test* (*DUT*). The testbench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced. The input and desired output patterns are called *test vectors*.

Consider testing the `sillyfunction` module from Section 4.1.1 that computes $y = \overline{a}\,\overline{b}\,\overline{c} + a\overline{b}\,\overline{c} + a\overline{b}c$. This is a simple module, so we can perform exhaustive testing by applying all eight possible test vectors.

HDL Example 4.37 TESTBENCH

SystemVerilog

```
module testbench1();
  logic a, b, c, y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
    a = 1; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
  end
endmodule
```

The `initial` statement executes the statements in its body at the start of simulation. In this case, it first applies the input pattern 000 and waits for 10 time units. It then applies 001 and waits 10 more units, and so forth until all eight possible inputs have been applied. `initial` statements should be used only in testbenches for simulation, not in modules intended to be synthesized into actual hardware. Hardware has no way of magically executing a sequence of special steps when it is first turned on.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench1 is -- no inputs or outputs
end;

architecture sim of testbench1 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
            y:      out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    wait; -- wait forever
  end process;
end;
```

The `process` statement first applies the input pattern 000 and waits for 10 ns. It then applies 001 and waits 10 more ns, and so forth until all eight possible inputs have been applied.

At the end, the process waits indefinitely; otherwise, the process would begin again, repeatedly applying the pattern of test vectors.

HDL Example 4.37 demonstrates a simple testbench. It instantiates the DUT, then applies the inputs. Blocking assignments and delays are used to apply the inputs in the appropriate order. The user must view the results of the simulation and verify by inspection that the correct outputs are produced. Testbenches are simulated the same as other HDL modules. However, they are not synthesizeable.

Checking for correct outputs is tedious and error-prone. Moreover, determining the correct outputs is much easier when the design is fresh in your mind; if you make minor changes and need to retest weeks later, determining the correct outputs becomes a hassle. A much better approach is to write a self-checking testbench, shown in HDL Example 4.38.

Writing code for each test vector also becomes tedious, especially for modules that require a large number of vectors. An even better approach

HDL Example 4.38 SELF-CHECKING TESTBENCH

SystemVerilog

```
module testbench2();
  logic a, b, c, y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  // checking results
  initial begin
    a = 0; b = 0; c = 0; #10;
    assert (y === 1) else $error("000 failed.");
    c = 1;  #10;
    assert (y === 0) else $error("001 failed.");
    b = 1; c = 0;  #10;
    assert (y === 0) else $error("010 failed.");
    c = 1;  #10;
    assert (y === 0) else $error("011 failed.");
    a = 1; b = 0; c = 0;  #10;
    assert (y === 1) else $error("100 failed.");
    c = 1;  #10;
    assert (y === 1) else $error("101 failed.");
    b = 1; c = 0;  #10;
    assert (y === 0) else $error("110 failed.");
    c = 1;  #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
```

The SystemVerilog `assert` statement checks if a specified condition is true. If not, it executes the `else` statement. The `$error` system task in the `else` statement prints an error message describing the assertion failure. `assert` is ignored during synthesis.

In SystemVerilog, comparison using == or != is effective between signals that do not take on the values of `x` and `z`. Testbenches use the === and !== operators for comparisons of equality and inequality, respectively, because these operators work correctly with operands that could be `x` or `z`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench2 is -- no inputs or outputs
end;

architecture sim of testbench2 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  -- checking results
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "000 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "001 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "010 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "011 failed.";
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "100 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '1' report "101 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "110 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "111 failed.";
    wait; -- wait forever
  end process;
end;
```

The `assert` statement checks a condition and prints the message given in the `report` clause if the condition is not satisfied. `assert` is meaningful only in simulation, not in synthesis.

is to place the test vectors in a separate file. The testbench simply reads the test vectors from the file, applies the input test vector to the DUT, waits, checks that the output values from the DUT match the output vector, and repeats until reaching the end of the test vectors file.

HDL Example 4.39 demonstrates such a testbench. The testbench generates a clock using an `always/process` statement with no sensitivity list, so that it is continuously reevaluated. At the beginning of the simulation, it reads the test vectors from a text file and pulses `reset` for two cycles. Although the clock and reset aren't necessary to test combinational logic, they are included because they would be important to testing a sequential DUT. `example.tv` is a text file containing the inputs and expected output written in binary:

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

HDL Example 4.39 TESTBENCH WITH TEST VECTOR FILE

SystemVerilog

```
module testbench3();
  logic         clk, reset;
  logic         a, b, c, y, yexpected;
  logic [31:0]  vectornum, errors;
  logic [3:0]   testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} = testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin  // check result
         $display("Error: inputs = %b", {a, b, c});
         $display(" outputs = %b (%b expected)", y, yexpected);
         errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
        $display("%d tests completed with %d errors",
                 vectornum, errors);
        $finish;
      end
    end
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_TEXTIO.ALL; use STD.TEXTIO.all;

entity testbench3 is -- no inputs or outputs
end;

architecture sim of testbench3 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y:  STD_LOGIC;
  signal y_expected: STD_LOGIC;
  signal clk, reset: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- at start of test, pulse reset
  process begin
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;

  -- run tests
  process is
    file tv: text;
    variable L: line;
    variable vector_in: std_logic_vector(2 downto 0);
    variable dummy: character;
    variable vector_out: std_logic;
    variable vectornum: integer := 0;
    variable errors: integer := 0;
  begin
    FILE_OPEN(tv, "example.tv", READ_MODE);
    while not endfile(tv) loop

      -- change vectors on rising edge
      wait until rising_edge(clk);

      -- read the next line of testvectors and split into pieces
      readline(tv, L);
      read(L, vector_in);
      read(L, dummy); -- skip over underscore
```

`$readmemb` reads a file of binary numbers into the `testvectors` array. `$readmemh` is similar but reads a file of hexadecimal numbers.

The next block of code waits one time unit after the rising edge of the clock (to avoid any confusion if clock and data change simultaneously), then sets the three inputs (`a`, `b`, and `c`) and the expected output (`yexpected`) based on the four bits in the current test vector.

The testbench compares the generated output, `y`, with the expected output, `yexpected`, and prints an error if they don't match. `%b` and `%d` indicate to print the values in binary and decimal, respectively. `$display` is a system task to print in the simulator window. For example, `$display("%b %b", y, yexpected);` prints the two values, `y` and `yexpected`, in binary. `%h` prints a value in hexadecimal.

This process repeats until there are no more valid test vectors in the `testvectors` array. `$finish` terminates the simulation.

Note that even though the SystemVerilog module supports up to 10,001 test vectors, it will terminate the simulation after executing the eight vectors in the file.

```
      read(L, vector_out);
      (a, b, c) <= vector_in(2 downto 0) after 1 ns;
      y_expected <= vector_out after 1 ns;

      -- check results on falling edge
      wait until falling_edge(clk);

      if y /= y_expected then
        report "Error: y = " & std_logic'image(y);
        errors := errors + 1;
      end if;

      vectornum := vectornum + 1;
    end loop;

    -- summarize results at end of simulation
    if (errors = 0) then
      report "NO ERRORS -- " &
             integer'image(vectornum) &
             " tests completed successfully."
             severity failure;
    else
      report integer'image(vectornum) &
             " tests completed, errors = " &
             integer'image(errors)
             severity failure;
    end if;
  end process;
end;
```

The VHDL code uses file reading commands beyond the scope of this chapter, but it gives the sense of what a self-checking testbench looks like in VHDL.

New inputs are applied on the rising edge of the clock, and the output is checked on the falling edge of the clock. Errors are reported as they occur. At the end of the simulation, the testbench prints the total number of test vectors applied and the number of errors detected.

The testbench in HDL Example 4.39 is overkill for such a simple circuit. However, it can easily be modified to test more complex circuits by changing the `example.tv` file, instantiating the new DUT, and changing a few lines of code to set the inputs and check the outputs.

4.10 SUMMARY

Hardware description languages (HDLs) are extremely important tools for modern digital designers. Once you have learned SystemVerilog or VHDL, you will be able to specify digital systems much faster than if you had to draw the complete schematics. The debug cycle is also often much faster, because modifications require code changes instead of tedious schematic rewiring. However, the debug cycle can be much *longer* using HDLs if you don't have a good idea of the hardware your code implies.

HDLs are used for both simulation and synthesis. Logic simulation is a powerful way to test a system on a computer before it is turned into hardware. Simulators let you check the values of signals inside your system that might be impossible to measure on a physical piece of hardware. Logic synthesis converts the HDL code into digital logic circuits.

The most important thing to remember when you are writing HDL code is that you are describing real hardware, not writing a computer program. The most common beginner's mistake is to write HDL code without thinking about the hardware you intend to produce. If you don't know what hardware you are implying, you are almost certain not to get what you want. Instead, begin by sketching a block diagram of your system, identifying which portions are combinational logic, which portions are sequential circuits or finite state machines, and so forth. Then write HDL code for each portion, using the correct idioms to imply the kind of hardware you need.

Exercises

The following exercises may be done using your favorite HDL. If you have a simulator available, test your design. Print the waveforms and explain how they prove that it works. If you have a synthesizer available, synthesize your code. Print the generated circuit diagram, and explain why it matches your expectations.

Exercise 4.1 Sketch a schematic of the circuit described by the following HDL code. Simplify the schematic so that it shows a minimum number of gates.

SystemVerilog

```
module exercise1(input  logic a, b, c,
                 output logic y, z);

  assign y = a & b & c | a & b & ~c | a & ~b & c;
  assign z = a & b | ~a & ~b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise1 is
  port(a, b, c: in   STD_LOGIC;
       y, z:    out  STD_LOGIC);
end;

architecture synth of exercise1 is
begin
  y <= (a and b and c) or (a and b and not c) or
       (a and not b and c);
  z <= (a and b) or (not a and not b);
end;
```

Exercise 4.2 Sketch a schematic of the circuit described by the following HDL code. Simplify the schematic so that it shows a minimum number of gates.

SystemVerilog

```
module exercise2(input  logic [3:0] a,
                 output logic [1:0] y);
  always_comb
    if      (a[0]) y = 2'b11;
    else if (a[1]) y = 2'b10;
    else if (a[2]) y = 2'b01;
    else if (a[3]) y = 2'b00;
    else           y = a[1:0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise2 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of exercise2 is
begin
  process(all) begin
    if    a(0) then y <= "11";
    elsif a(1) then y <= "10";
    elsif a(2) then y <= "01";
    elsif a(3) then y <= "00";
    else            y <= a(1 downto 0);
    end if;
  end process;
end;
```

Exercise 4.3 Write an HDL module that computes a four-input XOR function. The input is $a_{3:0}$, and the output is y.

Exercise 4.4 Write a self-checking testbench for Exercise 4.3. Create a test vector file containing all 16 test cases. Simulate the circuit and show that it works.

Introduce an error in the test vector file and show that the testbench reports a mismatch.

Exercise 4.5 Write an HDL module called `minority`. It receives three inputs, `a`, `b`, and `c`. It produces one output, `y`, that is TRUE if at least two of the inputs are FALSE.

Exercise 4.6 Write an HDL module for a hexadecimal seven-segment display decoder. The decoder should handle the digits A, B, C, D, E, and F as well as 0–9.

Exercise 4.7 Write a self-checking testbench for Exercise 4.6. Create a test vector file containing all 16 test cases. Simulate the circuit and show that it works. Introduce an error in the test vector file and show that the testbench reports a mismatch.

Exercise 4.8 Write an 8:1 multiplexer module called `mux8` with inputs $s_{2:0}$, `d0`, `d1`, `d2`, `d3`, `d4`, `d5`, `d6`, `d7`, and output `y`.

Exercise 4.9 Write a structural module to compute the logic function, $y = a\overline{b} + \overline{b}\,\overline{c} + \overline{a}bc$, using multiplexer logic. Use the 8:1 multiplexer from Exercise 4.8.

Exercise 4.10 Repeat Exercise 4.9 using a 4:1 multiplexer and as many NOT gates as you need.

Exercise 4.11 Section 4.5.4 pointed out that a synchronizer could be correctly described with blocking assignments if the assignments were given in the proper order. Think of a simple sequential circuit that cannot be correctly described with blocking assignments, regardless of order.

Exercise 4.12 Write an HDL module for an eight-input priority circuit.

Exercise 4.13 Write an HDL module for a 2:4 decoder.

Exercise 4.14 Write an HDL module for a 6:64 decoder using three instances of the 2:4 decoders from Exercise 4.13 and a bunch of three-input AND gates.

Exercise 4.15 Write HDL modules that implement the Boolean equations from Exercise 2.13.

Exercise 4.16 Write an HDL module that implements the circuit from Exercise 2.26.

Exercise 4.17 Write an HDL module that implements the circuit from Exercise 2.27.

Exercise 4.18 Write an HDL module that implements the logic function from Exercise 2.28. Pay careful attention to how you handle don't cares.

Exercise 4.19 Write an HDL module that implements the functions from Exercise 2.35.

Exercise 4.20 Write an HDL module that implements the priority encoder from Exercise 2.36.

Exercise 4.21 Write an HDL module that implements the modified priority encoder from Exercise 2.37.

Exercise 4.22 Write an HDL module that implements the binary-to-thermometer code converter from Exercise 2.38.

Exercise 4.23 Write an HDL module implementing the days-in-month function from Question 2.2.

Exercise 4.24 Sketch the state transition diagram for the FSM described by the following HDL code.

SystemVerilog

```
module fsm2(input  logic clk, reset,
            input  logic a, b,
            output logic y);

  logic [1:0] state, nextstate;

  parameter  S0  =  2'b00;
  parameter  S1  =  2'b01;
  parameter  S2  =  2'b10;
  parameter  S3  =  2'b11;

  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  always_comb
    case (state)
      S0: if (a ^ b) nextstate = S1;
          else       nextstate = S0;
      S1: if (a & b) nextstate = S2;
          else       nextstate = S0;
      S2: if (a | b) nextstate = S3;
          else       nextstate = S0;
      S3: if (a | b) nextstate = S3;
          else       nextstate = S0;
    endcase

  assign y = (state == S1) | (state == S2);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm2 is
  port(clk, reset: in  STD_LOGIC;
       a, b:       in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of fsm2 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  process(all) begin
    case state is
      when S0 => if (a xor b) then
                      nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if (a and b) then
                      nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if (a or b) then
                      nextstate <= S3;
                 else nextstate <= S0;
                 end if;
      when S3 => if (a or b) then
                      nextstate <= S3;
                 else nextstate <= S0;
                 end if;
    end case;
  end process;

  y <= '1' when ((state = S1) or (state = S2))
       else '0';
end;
```

Exercise 4.25 Sketch the state transition diagram for the FSM described by the following HDL code. An FSM of this nature is used in a branch predictor on some microprocessors.

SystemVerilog

```
module fsm1(input  logic clk, reset,
            input  logic taken, back,
            output logic predicttaken);

  logic [4:0] state, nextstate;

  parameter  S0  =  5'b00001;
  parameter  SI  =  5'b00010;
  parameter  S2  =  5'b00100;
  parameter  S3  =  5'b01000;
  parameter  S4  =  5'b10000;

  always_ff @(posedge clk, posedge  reset)
    if (reset) state <= S2;
    else       state <= nextstate;

  always_comb
    case (state)
      S0: if (taken)  nextstate = S1;
          else        nextstate = S0;
      S1: if (taken)  nextstate = S2;
          else        nextstate = S0;
      S2: if (taken)  nextstate = S3;
          else        nextstate = S1;
      S3: if (taken)  nextstate = S4;
          else        nextstate = S2;
      S4: if (taken)  nextstate = S4;
          else        nextstate = S3;
      default:        nextstate = S2;
    endcase

  assign predicttaken = (state  ==  S4)  |
                        (state  ==  S3)  |
                        (state  ==  S2 && back);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164. all;

entity fsm1 is
  port(clk, reset:     in   STD_LOGIC;
       taken, back:    in   STD_LOGIC;
       predicttaken:   out  STD_LOGIC);
end;

architecture synth of fsm1 is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S2;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  process(all) begin
    case state is
      when S0 => if taken then
                         nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if taken then
                         nextstate => S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if taken then
                         nextstate <= S3;
                 else nextstate <= S1;
                 end if;
      when S3 => if taken then
                         nextstate <= S4;
                 else nextstate <= S2;
                 end if;
      when S4 => if taken then
                         nextstate <= S4;
                 else nextstate <= S3;
                 end if;
      when others =>  nextstate <= S2;
    end case;
  end process;

  -- output logic
  predicttaken <= '1' when
                  ((state = S4) or (state = S3) or
                  (state = S2 and back = '1'))
  else '0';
end;
```

Exercise 4.26 Write an HDL module for an SR latch.

Exercise 4.27 Write an HDL module for a *JK flip-flop*. The flip-flop has inputs, *clk*, *J*, and *K*, and output *Q*. On the rising edge of the clock, *Q* keeps its old value if $J = K = 0$. It sets *Q* to 1 if $J = 1$, resets *Q* to 0 if $K = 1$, and inverts *Q* if $J = K = 1$.

Exercise 4.28 Write an HDL module for the latch from Figure 3.18. Use one assignment statement for each gate. Specify delays of 1 unit or 1 ns to each gate. Simulate the latch and show that it operates correctly. Then increase the inverter delay. How long does the delay have to be before a race condition causes the latch to malfunction?

Exercise 4.29 Write an HDL module for the traffic light controller from Section 3.4.1.

Exercise 4.30 Write three HDL modules for the factored parade mode traffic light controller from Example 3.8. The modules should be called `controller`, `mode`, and `lights`, and they should have the inputs and outputs shown in Figure 3.33(b).

Exercise 4.31 Write an HDL module describing the circuit in Figure 3.42.

Exercise 4.32 Write an HDL module for the FSM with the state transition diagram given in Figure 3.69 from Exercise 3.22.

Exercise 4.33 Write an HDL module for the FSM with the state transition diagram given in Figure 3.70 from Exercise 3.23.

Exercise 4.34 Write an HDL module for the improved traffic light controller from Exercise 3.24.

Exercise 4.35 Write an HDL module for the daughter snail from Exercise 3.25.

Exercise 4.36 Write an HDL module for the soda machine dispenser from Exercise 3.26.

Exercise 4.37 Write an HDL module for the Gray code counter from Exercise 3.27.

Exercise 4.38 Write an HDL module for the UP/DOWN Gray code counter from Exercise 3.28.

Exercise 4.39 Write an HDL module for the FSM from Exercise 3.29.

Exercise 4.40 Write an HDL module for the FSM from Exercise 3.30.

Exercise 4.41 Write an HDL module for the serial two's complementer from Question 3.2.

Exercise 4.42 Write an HDL module for the circuit in Exercise 3.31.

Exercise 4.43 Write an HDL module for the circuit in Exercise 3.32.

Exercise 4.44 Write an HDL module for the circuit in Exercise 3.33.

Exercise 4.45 Write an HDL module for the circuit in Exercise 3.34. You may use the full adder from Section 4.2.5.

SystemVerilog Exercises

The following exercises are specific to SystemVerilog.

Exercise 4.46 What does it mean for a signal to be declared `tri` in SystemVerilog?

Exercise 4.47 Rewrite the `syncbad` module from HDL Example 4.29. Use nonblocking assignments, but change the code to produce a correct synchronizer with two flip-flops.

Exercise 4.48 Consider the following two SystemVerilog modules. Do they have the same function? Sketch the hardware each one implies.

```
module code1(input  logic clk, a, b, c,
             output logic y);
  logic x;

  always_ff @(posedge clk) begin
    x <= a & b;
    y <= x | c;
  end
endmodule

module code2 (input  logic a, b, c, clk,
              output logic y);
  logic x;

  always_ff @(posedge clk) begin
    y <= x | c;
    x <= a & b;
  end
endmodule
```

Exercise 4.49 Repeat Exercise 4.48 if the <= is replaced by = in every assignment.

Exercise 4.50 The following SystemVerilog modules show errors that the authors have seen students make in the laboratory. Explain the error in each module and show how to fix it.

(a)
```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output reg   [3:0] q);

  always @(clk)
    if (clk) q <= d;
endmodule
```

(b)
```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);

  always @(a)
    begin
      y1 = a & b;
      y2 = a | b;
      y3 = a ^ b;
      y4 = ~(a & b);
      y5 = ~(a | b);
    end
endmodule
```

(c)
```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

    always @(posedge s)
      if (s) y <= d1;
      else   y <= d0;
endmodule
```

(d)
```
module twoflops(input  logic clk,
                input  logic d0, d1,
                output logic q0, q1);

    always @(posedge clk)
      q1 = d1;
      q0 = d0;
endmodule
```

(e)
```
module FSM(input  logic clk,
           input  logic a,
           output logic out1, out2);

    logic state;

    // next state logic and register (sequential)
    always_ff @(posedge clk)
      if (state == 0) begin
          if (a) state  <= 1;
      end else begin
          if (~a) state <= 0;
      end
    always_comb // output logic (combinational)
      if (state == 0)  out1 = 1;
      else             out2 = 1;
endmodule
```

(f)
```
module priority(input  logic [3:0] a,
                output logic [3:0] y);

    always_comb
      if      (a[3]) y = 4'b1000;
      else if (a[2]) y = 4'b0100;
      else if (a[1]) y = 4'b0010;
      else if (a[0]) y = 4'b0001;
endmodule
```

(g)
```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic out);

    logic [1:0] state, nextstate;

    parameter  S0  =  2'b00;
    parameter  S1  =  2'b01;
    parameter  S2  =  2'b10;

    // State Register
    always_ff @(posedge clk, posedge reset)
      if (reset)  state <= S0;
      else        state <= nextstate;

    // Next State Logic
    always @(state)
      case (state)
        S0: nextstate  =  S1;
        S1: nextstate  =  S2;
        S2: nextstate  =  S0;
      endcase

    // Output Logic
    assign out = (state == S2);
endmodule
```

(h)
```
module mux2tri(input  logic [3:0] d0, d1,
               input  logic       s,
               output tri   [3:0] y);
    tristate t0(d0, s, y);
    tristate t1(d1, s, y);
endmodule
```

(i)
```
module floprsen(input  logic       clk,
                input  logic       reset,
                input  logic       set,
                input  logic [3:0] d,
                output logic [3:0] q);
```

```
    always_ff @(posedge clk, posedge reset)
      if (reset) q <= 0;
      else       q <= d;

    always @(set)
      if (set)  q <= 1;
endmodule
```

(j)
```
module and3(input  logic a, b, c,
            output logic y);

    logic tmp;

    always @(a, b, c)
    begin
      tmp <= a & b;
      y   <= tmp & c;
    end
endmodule
```

VHDL Exercises

The following exercises are specific to VHDL.

Exercise 4.51 In VHDL, why is it necessary to write

```
q <= '1' when state = S0 else '0';
```

rather than simply

```
q <= (state = S0);
```

Exercise 4.52 Each of the following VHDL modules contains an error. For brevity, only the architecture is shown; assume that the `library` use clause and `entity` declaration are correct. Explain the error and show how to fix it.

(a)
```
architecture synth of latch is
begin
  process(clk) begin
    if clk = '1' then q <= d;
    end if;
  end process;
end;
```

(b)
```
architecture proc of gates is
begin
  process(a) begin
    Y1 <= a and b;
    y2 <= a or b;
    y3 <= a xor b;
    y4 <= a nand b;
    y5 <= a nor b;
  end process;
end;
```

(c)
```
architecture synth of flop is
begin
  process(clk)
    if rising_edge(clk) then
      q <= d;
end;
```

(d)
```
architecture synth of priority is
begin
  process(all) begin
    if    a(3) then y <= "1000";
    elsif a(2) then y <= "0100";
    elsif a(1) then y <= "0010";
    elsif a(0) then y <= "0001";
    end if;
  end process;
end;
```

(e)
```
architecture synth of divideby3FSM is
    type statetype is (S0, S1, S2);
    signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset then state <= S0;
    elsif rising_edge(clk) then
      state <= nextstate;
    end if;
  end process;

  process(state) begin
    case state is
      when S0 => nextstate <= S1;
      when S1 => nextstate <= S2;
      when S2 => nextstate <= S0;
    end case;
  end process;

  q <= '1' when state = S0 else '0';
end;
```

(f)
```
architecture struct of mux2 is
   component tristate
     port(a:  in   STD_LOGIC_VECTOR(3 downto 0);
          en: in   STD_LOGIC;
          y:  out  STD_LOGIC_VECTOR(3 downto 0));
   end component;

begin
  t0: tristate port map(d0, s, y);
  t1: tristate port map(d1, s, y);
end;
```

(g)
```
architecture asynchronous of floprs is
begin
  process(clk, reset) begin
    if reset then
      q <= '0';
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;

  process(set) begin
    if set then
      q <= '1';
    end if;
  end process;
end;
```

Interview Questions

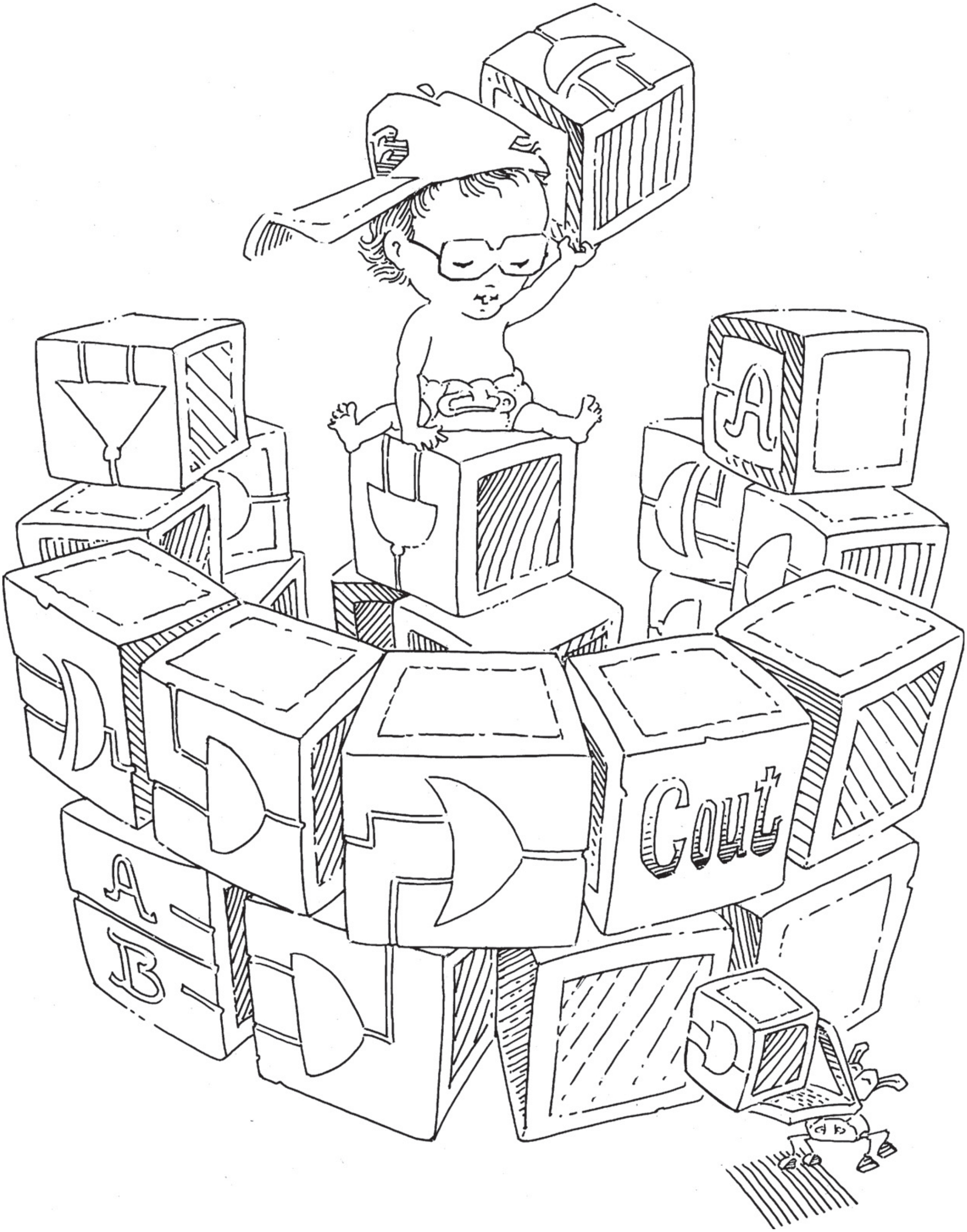
The following exercises present questions that have been asked at interviews for digital design jobs.

Question 4.1 Write a line of HDL code that gates a 32-bit bus called `data` with another signal called `sel` to produce a 32-bit `result`. If `sel` is TRUE, `result` = `data`. Otherwise, `result` should be all 0's.

Question 4.2 Explain the difference between blocking and nonblocking assignments in SystemVerilog. Give examples.

Question 4.3 What does the following SystemVerilog statement do?

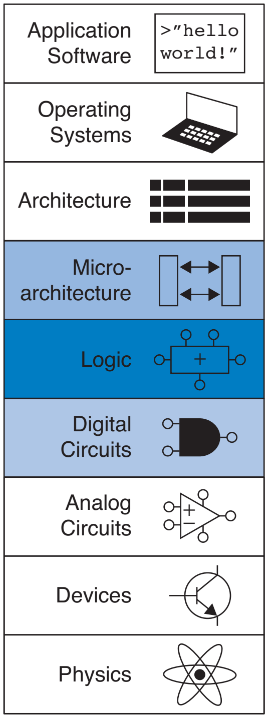
```
result = | (data[15:0] & 16'hC820);
```


A
A
B
Cout

5

Digital Building Blocks

5.1 Introduction
5.2 Arithmetic Circuits
5.3 Number Systems
5.4 Sequential Building Blocks
5.5 Memory Arrays
5.6 Logic Arrays
5.7 Summary
Exercises
Interview Questions



5.1 INTRODUCTION

Up to this point, we have examined the design of combinational and sequential circuits using Boolean equations, schematics, and HDLs. This chapter introduces more elaborate combinational and sequential building blocks used in digital systems. These blocks include arithmetic circuits, counters, shift registers, memory arrays, and logic arrays. These building blocks are not only useful in their own right, but they also demonstrate the principles of hierarchy, modularity, and regularity. The building blocks are hierarchically assembled from simpler components such as logic gates, multiplexers, and decoders. Each building block has a well-defined interface and can be treated as a black box when the underlying implementation is unimportant. The regular structure of each building block is easily extended to different sizes. In Chapter 7, we use many of these building blocks to build a microprocessor.

5.2 ARITHMETIC CIRCUITS

Arithmetic circuits are the central building blocks of computers. Computers and digital logic perform many arithmetic functions: addition, subtraction, comparisons, shifts, multiplication, and division. This section describes hardware implementations for all of these operations.

5.2.1 Addition

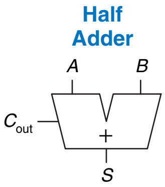
Addition is one of the most common operations in digital systems. We first consider how to add two 1-bit binary numbers. We then extend to *N*-bit binary numbers. Adders also illustrate trade-offs between speed and complexity.

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00005-7
© 2013 Elsevier, Inc. All rights reserved.

Half Adder

We begin by building a 1-bit *half adder*. As shown in Figure 5.1, the half adder has two inputs, A and B, and two outputs, S and C_{out}. S is the sum of A and B. If A and B are both 1, S is 2, which cannot be represented with a single binary digit. Instead, it is indicated with a carry out C_{out} in the next column. The half adder can be built from an XOR gate and an AND gate.

In a multi-bit adder, C_{out} is added or *carried in* to the next most significant bit. For example, in Figure 5.2, the carry bit shown in blue is the output C_{out} of the first column of 1-bit addition and the input C_{in} to the second column of addition. However, the half adder lacks a C_{in} input to accept C_{out} of the previous column. The *full adder*, described in the next section, solves this problem.



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$
$$C_{out} = AB$$

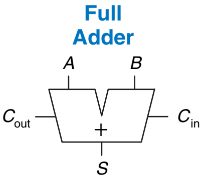
Figure 5.1 **1-bit half adder**

$$\begin{array}{r} 1 \\ 0001 \\ +0101 \\ \hline 0110 \end{array}$$

Figure 5.2 **Carry bit**

Full Adder

A *full adder*, introduced in Section 2.1, accepts the carry in C_{in} as shown in Figure 5.3. The figure also shows the output equations for S and C_{out}.



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Figure 5.3 **1-bit full adder**

Carry Propagate Adder

An N-bit adder sums two N-bit inputs, A and B, and a carry in C_{in} to produce an N-bit result S and a carry out C_{out}. It is commonly called a *carry propagate adder* (*CPA*) because the carry out of one bit propagates into the next bit. The symbol for a CPA is shown in Figure 5.4; it is drawn just like a full adder except that A, B, and S are busses rather than single bits. Three common CPA implementations are called ripple-carry adders, carry-lookahead adders, and prefix adders.

Ripple-Carry Adder

The simplest way to build an N-bit carry propagate adder is to chain together N full adders. The C_{out} of one stage acts as the C_{in} of the next stage, as shown in Figure 5.5 for 32-bit addition. This is called a *ripple-carry adder*. It is a good application of modularity and regularity: the full adder module is reused many times to form a larger system. The ripple-carry adder has the disadvantage of being slow when N is large. S_{31} depends on C_{30}, which depends on C_{29}, which depends on C_{28}, and so forth all the way back to C_{in}, as shown in blue in Figure 5.5. We say that the carry *ripples* through the carry chain. The delay of the adder, t_{ripple},

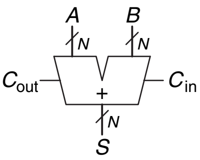


Figure 5.4 **Carry propagate adder**

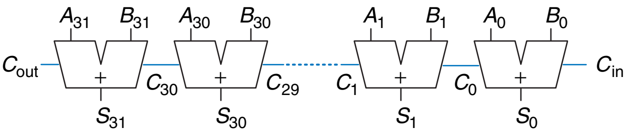


Figure 5.5 **32-bit ripple-carry adder**

grows directly with the number of bits, as given in Equation 5.1, where t_{FA} is the delay of a full adder.

$$t_{\text{ripple}} = Nt_{FA} \tag{5.1}$$

Carry-Lookahead Adder

The fundamental reason that large ripple-carry adders are slow is that the carry signals must propagate through every bit in the adder. A *carry-lookahead* adder (*CLA*) is another type of carry propagate adder that solves this problem by dividing the adder into *blocks* and providing circuitry to quickly determine the carry out of a block as soon as the carry in is known. Thus it is said to *look ahead* across the blocks rather than waiting to ripple through all the full adders inside a block. For example, a 32-bit adder may be divided into eight 4-bit blocks.

CLAs use *generate* (G) and *propagate* (P) signals that describe how a column or block determines the carry out. The ith column of an adder is said to *generate* a carry if it produces a carry out independent of the carry in. The ith column of an adder is guaranteed to generate a carry C_i if A_i and B_i are both 1. Hence G_i, the generate signal for column i, is calculated as $G_i = A_iB_i$. The column is said to *propagate* a carry if it produces a carry out whenever there is a carry in. The ith column will propagate a carry in, C_{i-1}, if either A_i or B_i is 1. Thus, $P_i = A_i + B_i$. Using these definitions, we can rewrite the carry logic for a particular column of the adder. The ith column of an adder will generate a carry out C_i if it either generates a carry, G_i, or propagates a carry in, P_iC_{i-1}. In equation form,

$$C_i = A_iB_i + (A_i + B_i)C_{i-1} = G_i + P_iC_{i-1} \tag{5.2}$$

The generate and propagate definitions extend to multiple-bit blocks. A block is said to generate a carry if it produces a carry out independent of the carry in to the block. The block is said to propagate a carry if it produces a carry out whenever there is a carry in to the block. We define $G_{i:j}$ and $P_{i:j}$ as generate and propagate signals for blocks spanning columns i through j.

A block generates a carry if the most significant column generates a carry, or if the most significant column propagates a carry and the previous column generated a carry, and so forth. For example, the generate logic for a block spanning columns 3 through 0 is

$$G_{3:0} = G_3 + P_3\big(G_2 + P_2(G_1 + P_1G_0)\big) \tag{5.3}$$

A block propagates a carry if all the columns in the block propagate the carry. For example, the propagate logic for a block spanning columns 3 through 0 is

$$P_{3:0} = P_3P_2P_1P_0 \tag{5.4}$$

Using the block generate and propagate signals, we can quickly compute the carry out of the block, C_i, using the carry in to the block, C_j.

$$C_i = G_{i:j} + P_{i:j}C_j \tag{5.5}$$

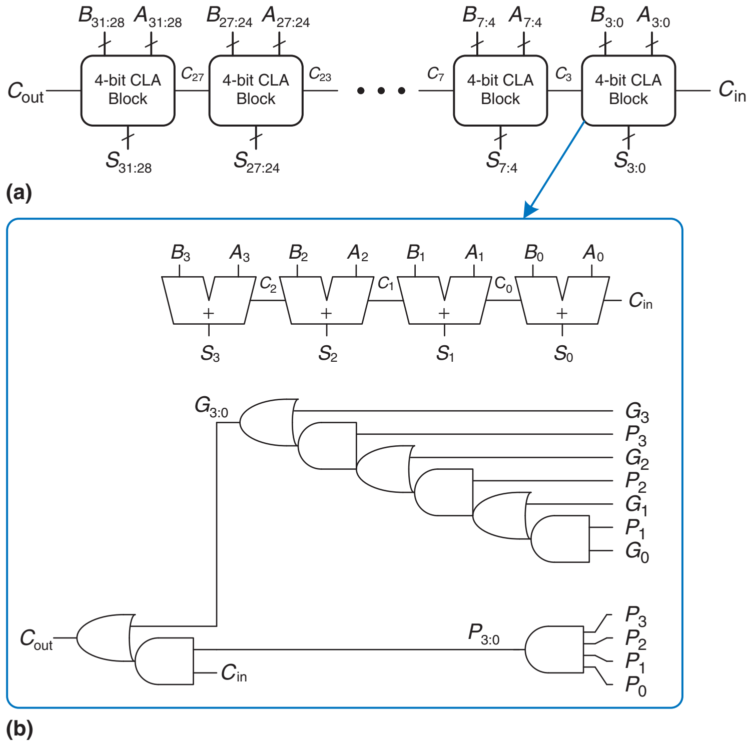
Schematics typically show signals flowing from left to right. Arithmetic circuits break this rule because the carries flow from right to left (from the least significant column to the most significant column).

Throughout the ages, people have used many devices to perform arithmetic. Toddlers count on their fingers (and some adults stealthily do too). The Chinese and Babylonians invented the abacus as early as 2400 BC. Slide rules, invented in 1630, were in use until the 1970's, when scientific hand calculators became prevalent. Computers and digital calculators are ubiquitous today. What will be next?

Figure 5.6(a) shows a 32-bit carry-lookahead adder composed of eight 4-bit blocks. Each block contains a 4-bit ripple-carry adder and some lookahead logic to compute the carry out of the block given the carry in, as shown in Figure 5.6(b). The AND and OR gates needed to compute the single-bit generate and propagate signals, G_i and P_i, from A_i and B_i are left out for brevity. Again, the carry-lookahead adder demonstrates modularity and regularity.

All of the CLA blocks compute the single-bit and block generate and propagate signals simultaneously. The critical path starts with computing G_0 and $G_{3:0}$ in the first CLA block. C_{in} then advances directly to C_{out} through the AND/OR gate in each block until the last. For a large adder, this is much faster than waiting for the carries to ripple through each consecutive bit of the adder. Finally, the critical path through the last block contains a short ripple-carry adder. Thus, an N-bit adder divided into k-bit blocks has a delay

$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + \left(\frac{N}{k} - 1\right) t_{\text{AND\_OR}} + k t_{FA} \tag{5.6}$$



$B_{31:28}$ $A_{31:28}$
$B_{27:24}$ $A_{27:24}$
$B_{7:4}$ $A_{7:4}$
$B_{3:0}$ $A_{3:0}$
C_{out}
4-bit CLA Block
C_{27}
4-bit CLA Block
C_{23}
C_7
4-bit CLA Block
C_3
4-bit CLA Block
C_{in}
$S_{31:28}$
$S_{27:24}$
$S_{7:4}$
$S_{3:0}$
(a)
B_3 A_3 B_2 A_2 B_1 A_1 B_0 A_0
C_2
C_1
C_0
C_{in}
+
S_3 S_2 S_1 S_0
$G_{3:0}$
G_3
P_3
G_2
P_2
G_1
P_1
G_0
C_{out}
$P_{3:0}$
C_{in}
P_3
P_2
P_1
P_0
(b)

Figure 5.6 **(a) 32-bit carry-lookahead adder (CLA), (b) 4-bit CLA block**

where t_{pg} is the delay of the individual generate/propagate gates (a single AND or OR gate) to generate P_i and G_i, t_{pg_block} is the delay to find the generate/propagate signals $P_{i:j}$ and $G_{i:j}$ for a k-bit block, and t_{AND_OR} is the delay from C_{in} to C_{out} through the final AND/OR logic of the k-bit CLA block. For $N > 16$, the carry-lookahead adder is generally much faster than the ripple-carry adder. However, the adder delay still increases linearly with N.

Example 5.1 RIPPLE-CARRY ADDER AND CARRY-LOOKAHEAD ADDER DELAY

Compare the delays of a 32-bit ripple-carry adder and a 32-bit carry-lookahead adder with 4-bit blocks. Assume that each two-input gate delay is 100 ps and that a full adder delay is 300 ps.

Solution: According to Equation 5.1, the propagation delay of the 32-bit ripple-carry adder is 32×300 ps $= 9.6$ ns.

The CLA has $t_{pg} = 100$ ps, $t_{pg\_block} = 6 \times 100$ ps $= 600$ ps, and $t_{AND\_OR} = 2 \times 100$ ps $=$ 200 ps. According to Equation 5.6, the propagation delay of the 32-bit carry-lookahead adder with 4-bit blocks is thus 100 ps + 600 ps + $(32/4 - 1) \times 200$ ps + $(4 \times 300$ ps$)$ = 3.3 ns, almost three times faster than the ripple-carry adder.

Prefix Adder*

Prefix adders extend the generate and propagate logic of the carry-lookahead adder to perform addition even faster. They first compute G and P for pairs of columns, then for blocks of 4, then for blocks of 8, then 16, and so forth until the generate signal for every column is known. The sums are computed from these generate signals.

In other words, the strategy of a prefix adder is to compute the carry in C_{i-1} for each column i as quickly as possible, then to compute the sum, using

$$S_i = (A_i \oplus B_i) \oplus C_{i-1} \tag{5.7}$$

Define column $i = -1$ to hold C_{in}, so $G_{-1} = C_{in}$ and $P_{-1} = 0$. Then $C_{i-1} = G_{i-1:-1}$ because there will be a carry out of column $i-1$ if the block spanning columns $i-1$ through -1 generates a carry. The generated carry is either generated in column $i-1$ or generated in a previous column and propagated. Thus, we rewrite Equation 5.7 as

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1} \tag{5.8}$$

Hence, the main challenge is to rapidly compute all the block generate signals $G_{-1:-1}, G_{0:-1}, G_{1:-1}, G_{2:-1}, \ldots, G_{N-2:-1}$. These signals, along with $P_{-1:-1}, P_{0:-1}, P_{1:-1}, P_{2:-1}, \ldots, P_{N-2:-1}$, are called *prefixes*.

Early computers used ripple-carry adders, because components were expensive and ripple-carry adders used the least hardware. Virtually all modern PCs use prefix adders on critical paths, because transistors are now cheap and speed is of great importance.

Figure 5.7 shows an $N = 16$-bit prefix adder. The adder begins with a *precomputation* to form P_i and G_i for each column from A_i and B_i using AND and OR gates. It then uses $\log_2 N = 4$ levels of black cells to form the prefixes of $G_{i:j}$ and $P_{i:j}$. A black cell takes inputs from the upper part of a block spanning bits *i:k* and from the lower part spanning bits *k*−1:*j*. It combines these parts to form generate and propagate signals for the entire block spanning bits *i:j* using the equations

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j} \tag{5.9}$$

$$P_{i:j} = P_{i:k} P_{k-1:j} \tag{5.10}$$

In other words, a block spanning bits *i:j* will generate a carry if the upper part generates a carry or if the upper part propagates a carry generated in the lower part. The block will propagate a carry if both the upper and

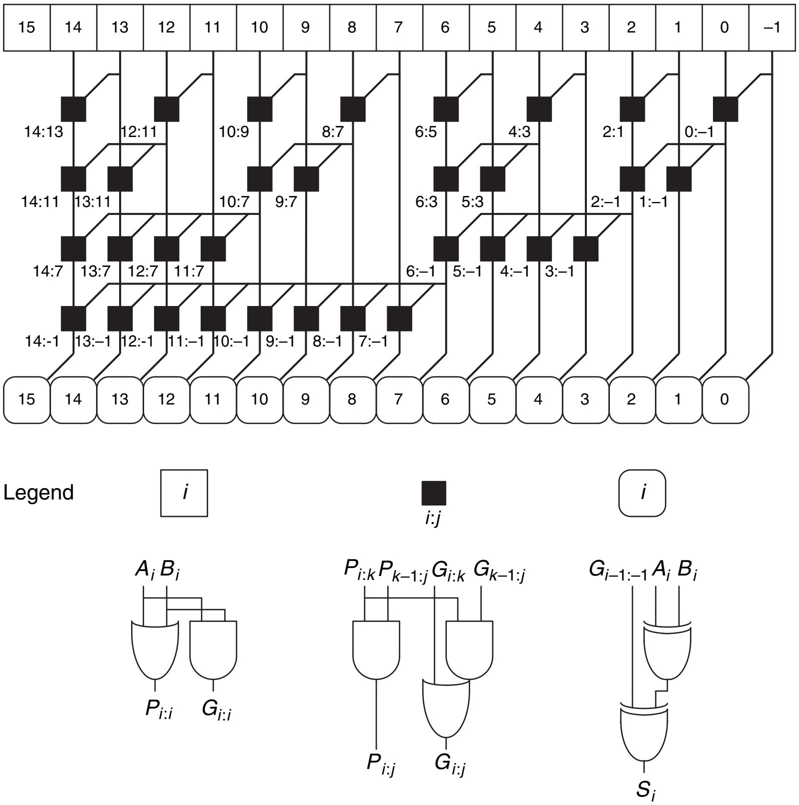

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
–1
14:13
12:11
10:9
8:7
6:5
4:3
2:1
0:–1
14:11
13:11
10:7
9:7
6:3
5:3
2:–1
1:–1
14:7
13:7
12:7
11:7
6:–1
5:–1
4:–1
3:–1
14:-1
13:–1
12:-1
11:–1
10:–1
9:–1
8:–1
7:–1
Legend
i
i:j
i
A_i B_i
$P_{i:i}$
$G_{i:i}$
$P_{i:k}$ $P_{k-1:j}$ $G_{i:k}$ $G_{k-1:j}$
$P_{i:j}$
$G_{i:j}$
$G_{i-1:-1}$ A_i B_i
S_i

Figure 5.7 **16-bit prefix adder**

lower parts propagate the carry. Finally, the prefix adder computes the sums using Equation 5.8.

In summary, the prefix adder achieves a delay that grows logarithmically rather than linearly with the number of columns in the adder. This speedup is significant, especially for adders with 32 or more bits, but it comes at the expense of more hardware than a simple carry-lookahead adder. The network of black cells is called a *prefix tree*.

The general principle of using prefix trees to perform computations in time that grows logarithmically with the number of inputs is a powerful technique. With some cleverness, it can be applied to many other types of circuits (see, for example, Exercise 5.7).

The critical path for an N-bit prefix adder involves the precomputation of P_i and G_i followed by $\log_2 N$ stages of black prefix cells to obtain all the prefixes. $G_{i\text{-}1:-1}$ then proceeds through the final XOR gate at the bottom to compute S_i. Mathematically, the delay of an N-bit prefix adder is

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}} \tag{5.11}$$

where t_{pg_prefix} is the delay of a black prefix cell.

Example 5.2 PREFIX ADDER DELAY

Compute the delay of a 32-bit prefix adder. Assume that each two-input gate delay is 100 ps.

Solution: The propagation delay of each black prefix cell t_{pg_prefix} is 200 ps (i.e., two gate delays). Thus, using Equation 5.11, the propagation delay of the 32-bit prefix adder is $100\text{ ps} + \log_2(32) \times 200\text{ ps} + 100\text{ ps} = 1.2\text{ ns}$, which is about three times faster than the carry-lookahead adder and eight times faster than the ripple-carry adder from Example 5.1. In practice, the benefits are not quite this great, but prefix adders are still substantially faster than the alternatives.

Putting It All Together

This section introduced the half adder, full adder, and three types of carry propagate adders: ripple-carry, carry-lookahead, and prefix adders. Faster adders require more hardware and therefore are more expensive and power-hungry. These trade-offs must be considered when choosing an appropriate adder for a design.

Hardware description languages provide the + operation to specify a CPA. Modern synthesis tools select among many possible implementations, choosing the cheapest (smallest) design that meets the speed requirements. This greatly simplifies the designer's job. HDL Example 5.1 describes a CPA with carries in and out.

HDL Example 5.1 ADDER

SystemVerilog

```
module adder #(parameter N = 8)
              (input  logic [N-1:0] a, b,
               input  logic         cin,
               output logic [N-1:0] s,
               output logic         cout);

  assign {cout, s} = a + b + cin;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity adder is
  generic(N: integer := 8);
  port(a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
       cin:  in  STD_LOGIC;
       s:    out STD_LOGIC_VECTOR(N-1 downto 0);
       cout: out STD_LOGIC);
end;

architecture synth of adder is
  signal result: STD_LOGIC_VECTOR(N downto 0);
begin
  result <= ("0" & a) + ("0" & b) + cin;
  s      <= result(N-1 downto 0);
  cout   <= result(N);
end;
```

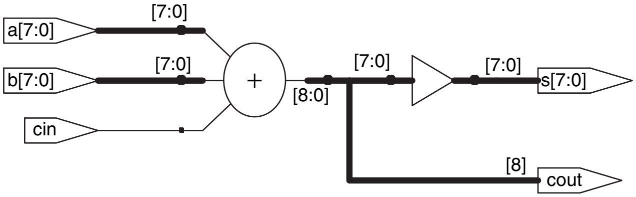


Figure 5.8 **Synthesized adder**

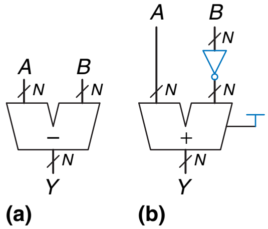


Figure 5.9 **Subtractor: (a) symbol, (b) implementation**

5.2.2 Subtraction

Recall from Section 1.4.6 that adders can add positive and negative numbers using two's complement number representation. Subtraction is almost as easy: flip the sign of the second number, then add. Flipping the sign of a two's complement number is done by inverting the bits and adding 1.

To compute $Y = A - B$, first create the two's complement of B: Invert the bits of B to obtain $\overline{B}$ and add 1 to get $-B = \overline{B} + 1$. Add this quantity to A to get $Y = A + \overline{B} + 1 = A - B$. This sum can be performed with a single CPA by adding $A + \overline{B}$ with $C_{\text{in}} = 1$. Figure 5.9 shows the symbol for a subtractor and the underlying hardware for performing $Y = A - B$. HDL Example 5.2 describes a subtractor.

5.2.3 Comparators

A *comparator* determines whether two binary numbers are equal or if one is greater or less than the other. A comparator receives two N-bit binary numbers A and B. There are two common types of comparators.

HDL Example 5.2 SUBTRACTOR

SystemVerilog

```
module subtractor #(parameter N = 8)
                   (input  logic [N-1:0] a, b,
                    output logic [N-1:0] y);

  assign y = a – b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity subtractor is
  generic(N: integer := 8);
  port(a, b: in  STD_LOGIC_VECTOR(N–1 downto 0);
       y:    out STD_LOGIC_VECTOR(N–1 downto 0));
end;

architecture synth of subtractor is
begin
  y <= a – b;
end;
```

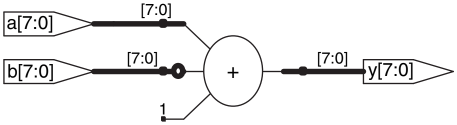


Figure 5.10 Synthesized subtractor

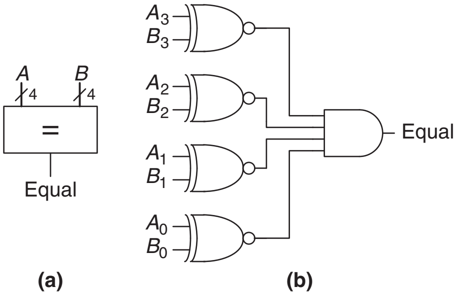


Figure 5.11 4-bit equality comparator: (a) symbol, (b) implementation

An *equality comparator* produces a single output indicating whether A is equal to B ($A == B$). A *magnitude comparator* produces one or more outputs indicating the relative values of A and B.

The equality comparator is the simpler piece of hardware. Figure 5.11 shows the symbol and implementation of a 4-bit equality comparator. It first checks to determine whether the corresponding bits in each column of A and B are equal using XNOR gates. The numbers are equal if all of the columns are equal.

Magnitude comparison is usually done by computing $A - B$ and looking at the sign (most significant bit) of the result as shown in Figure 5.12. If the result is negative (i.e., the sign bit is 1), then A is less than B. Otherwise A is greater than or equal to B.

HDL Example 5.3 shows how to use various comparison operations.

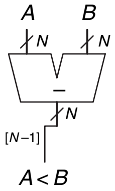


Figure 5.12 *N*-bit magnitude comparator

HDL Example 5.3 COMPARATORS

SystemVerilog

```
module comparator #(parameter N = 8)
                   (input  logic [N-1:0] a, b,
                    output logic eq, neq, lt, lte, gt, gte);

  assign eq  = (a == b);
  assign neq = (a != b);
  assign lt  = (a < b);
  assign lte = (a <= b);
  assign gt  = (a > b);
  assign gte = (a >= b);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;

entity comparators is
  generic(N: integer : = 8);
  port(a, b: in STD_LOGIC_VECTOR(N-1 downto 0);
       eq, neq, lt, lte, gt, gte: out STD_LOGIC);
end;

architecture synth of comparator is
begin
  eq  <= '1' when (a = b)  else '0';
  neq <= '1' when (a /= b) else '0';
  lt  <= '1' when (a < b)  else '0';
  lte <= '1' when (a <= b) else '0';
  gt  <= '1' when (a > b)  else '0';
  gte <= '1' when (a >= b) else '0';
end;
```

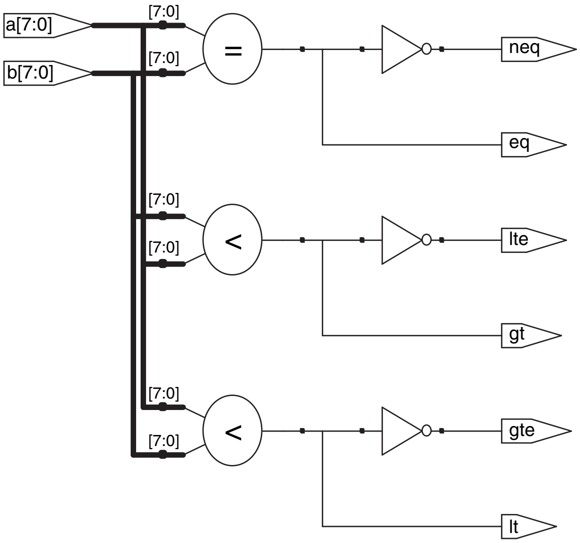


Figure 5.13 **Synthesized comparators**

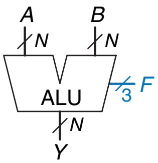


Figure 5.14 **ALU symbol**

5.2.4 ALU

An *Arithmetic/Logical Unit* (*ALU*) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. The ALU forms the heart of most computer systems.

Figure 5.14 shows the symbol for an *N*-bit ALU with *N*-bit inputs and outputs. The ALU receives a control signal *F* that specifies which

Table 5.1 ALU operations

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{\text{B}}$
101	A OR $\overline{\text{B}}$
110	A – B
111	SLT

function to perform. Control signals will generally be shown in blue to distinguish them from the data. Table 5.1 lists typical functions that the ALU can perform. The SLT function is used for magnitude comparison and will be discussed later in this section.

Figure 5.15 shows an implementation of the ALU. The ALU contains an N-bit adder and N two-input AND and OR gates. It also contains inverters and a multiplexer to invert input B when the F_2 control signal is asserted. A 4:1 multiplexer chooses the desired function based on the $F_{1:0}$ control signals.

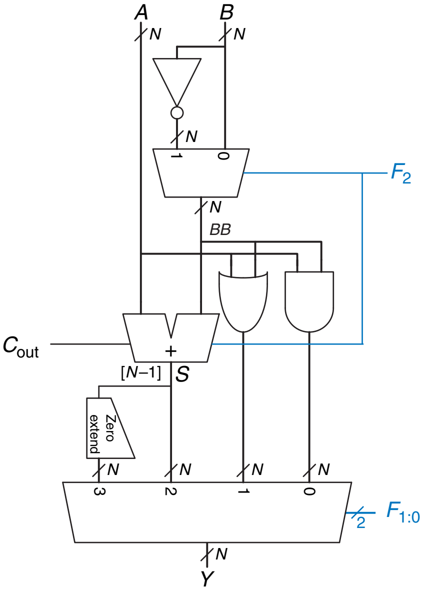


Figure 5.15 ***N*-bit ALU**

More specifically, the arithmetic and logical blocks in the ALU operate on A and BB. BB is either B or $\overline{B}$, depending on F_2. If $F_{1:0} = 00$, the output multiplexer chooses A AND BB. If $F_{1:0} = 01$, the ALU computes A OR BB. If $F_{1:0} = 10$, the ALU performs addition or subtraction. Note that F_2 is also the carry in to the adder. Also remember that $\overline{B} + 1 = -B$ in two's complement arithmetic. If $F_2 = 0$, the ALU computes $A + B$. If $F_2 = 1$, the ALU computes $A + \overline{B} + 1 = A - B$.

When $F_{2:0} = 111$, the ALU performs the *set if less than* (*SLT*) operation. When $A < B$, $Y = 1$. Otherwise, $Y = 0$. In other words, Y is set to 1 if A is less than B.

SLT is performed by computing $S = A - B$. If S is negative (i.e., the sign bit is set), A is less than B. The *zero extend unit* produces an N-bit output by concatenating its 1-bit input with 0's in the most significant bits. The sign bit (the N-1th bit) of S is the input to the zero extend unit.

Example 5.3 SET LESS THAN

Configure a 32-bit ALU for the SLT operation. Suppose $A = 25_{10}$ and $B = 32_{10}$. Show the control signals and output, Y.

Solution: Because $A < B$, we expect Y to be 1. For SLT, $F_{2:0} = 111$. With $F_2 = 1$, this configures the adder unit as a subtractor with an output S of $25_{10} - 32_{10} = -7_{10} = 1111 \ldots 1001_2$. With $F_{1:0} = 11$, the final multiplexer sets $Y = S_{31} = 1$.

Some ALUs produce extra outputs, called *flags*, that indicate information about the ALU output. For example, an *overflow flag* indicates that the result of the adder overflowed. A *zero flag* indicates that the ALU output is 0.

The HDL for an N-bit ALU is left to Exercise 5.9. There are many variations on this basic ALU that support other functions, such as XOR or equality comparison.

5.2.5 Shifters and Rotators

Shifters and *rotators* move bits and multiply or divide by powers of 2. As the name implies, a shifter shifts a binary number left or right by a specified number of positions. There are several kinds of commonly used shifters:

- **Logical shifter**—shifts the number to the left (LSL) or right (LSR) and fills empty spots with 0's.

  Ex: 11001 LSR 2 = 00110; 11001 LSL 2 = 00100

- **Arithmetic shifter**—is the same as a logical shifter, but on right shifts fills the most significant bits with a copy of the old most significant bit (msb). This is useful for multiplying and dividing signed numbers (see Sections 5.2.6 and 5.2.7). Arithmetic shift left (ASL) is the same as logical shift left (LSL).

Ex: 11001 ASR 2 = 11110; 11001 ASL 2 = 00100

- **Rotator**—rotates number in circle such that empty spots are filled with bits shifted off the other end.

  Ex: 11001 ROR 2 = 01110; 11001 ROL 2 = 00111

An *N*-bit shifter can be built from *N N*:1 multiplexers. The input is shifted by 0 to $N-1$ bits, depending on the value of the $\log_2 N$-bit select lines. Figure 5.16 shows the symbol and hardware of 4-bit shifters. The operators <<, >>, and >>> typically indicate shift left, logical shift right, and arithmetic shift right, respectively. Depending on the value of the 2-bit shift amount $shamt_{1:0}$, the output Y receives the input A shifted by 0 to 3 bits. For all shifters, when $shamt_{1:0} = 00$, $Y = A$. Exercise 5.14 covers rotator designs.

A left shift is a special case of multiplication. A left shift by N bits multiplies the number by 2^N. For example, $000011_2 << 4 = 110000_2$ is equivalent to $3_{10} \times 2^4 = 48_{10}$.

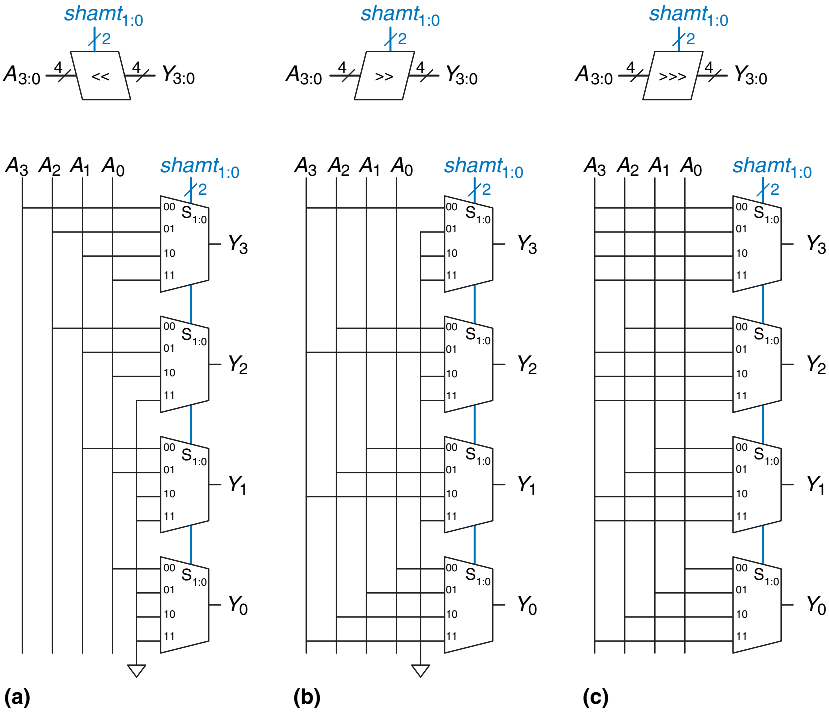


Figure 5.16 4-bit shifters: (a) shift left, (b) logical shift right, (c) arithmetic shift right

An arithmetic right shift is a special case of division. An arithmetic right shift by N bits divides the number by 2^N. For example, 11100_2 >>> $2 = 11111_2$ is equivalent to $-4_{10}/2^2 = -1_{10.}$

5.2.6 Multiplication*

Multiplication of unsigned binary numbers is similar to decimal multiplication but involves only 1's and 0's. Figure 5.17 compares multiplication in decimal and binary. In both cases, *partial products* are formed by multiplying a single digit of the multiplier with the entire multiplicand. The shifted partial products are summed to form the result.

In general, an $N \times N$ multiplier multiplies two N-bit numbers and produces a $2N$-bit result. The partial products in binary multiplication are either the multiplicand or all 0's. Multiplication of 1-bit binary numbers is equivalent to the AND operation, so AND gates are used to form the partial products.

Figure 5.18 shows the symbol, function, and implementation of a 4 × 4 multiplier. The multiplier receives the multiplicand and multiplier, A and B, and produces the product P. Figure 5.18(b) shows how partial products are formed. Each partial product is a single multiplier bit (B_3, B_2, B_1, or B_0) AND the multiplicand bits (A_3, A_2, A_1, A_0). With N-bit operands, there

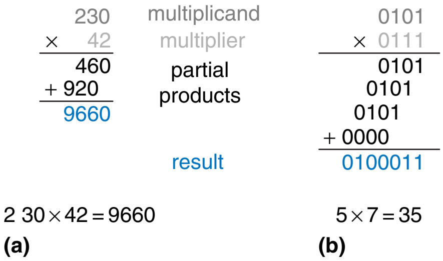

230
× 42
460
+ 920
9660
multiplicand
multiplier
partial products
result
0101
× 0111
0101
0101
0101
+ 0000
0100011
2 30×42=9660
5×7=35
(a)
(b)

Figure 5.17 Multiplication: (a) decimal, (b) binary

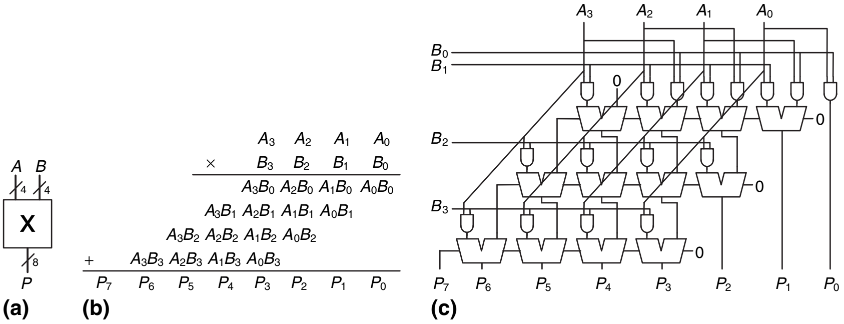

A
B
4
4
X
8
P
(a)
A3 A2 A1 A0
× B3 B2 B1 B0
A3B0 A2B0 A1B0 A0B0
A3B1 A2B1 A1B1 A0B1
A3B2 A2B2 A1B2 A0B2
+ A3B3 A2B3 A1B3 A0B3
P7 P6 P5 P4 P3 P2 P1 P0
(b)
A3 A2 A1 A0
B0
B1
B2
B3
0
0
0
0
P7 P6 P5 P4 P3 P2 P1 P0
(c)

Figure 5.18 4 × 4 multiplier: (a) symbol, (b) function, (c) implementation

are N partial products and $N-1$ stages of 1-bit adders. For example, for a 4×4 multiplier, the partial product of the first row is B_0 AND (A_3, A_2, A_1, A_0). This partial product is added to the shifted second partial product, B_1 AND (A_3, A_2, A_1, A_0). Subsequent rows of AND gates and adders form and add the remaining partial products.

The HDL for a multiplier is in HDL Example 5.4. As with adders, many different multiplier designs with different speed/cost trade-offs exist. Synthesis tools may pick the most appropriate design given the timing constraints.

HDL Example 5.4 MULTIPLIER

SystemVerilog

```
module multiplier #(parameter N = 8)
                   (input  logic [N-1:0]   a, b,
                    output logic [2*N-1:0] y);

  assign y = a * b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity multiplier is
  generic(N: integer := 8);
  port(a, b: in  STD_LOGIC_VECTOR(N-1 downto 0);
       y:    out STD_LOGIC_VECTOR(2*N-1 downto 0));
end;

architecture synth of multiplier is
begin
  y <= a * b;
end;
```

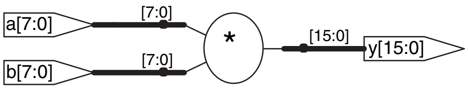


Figure 5.19 Synthesized multiplier

5.2.7 Division*

Binary division can be performed using the following algorithm for N-bit unsigned numbers in the range $[0, 2^{N-1}]$:

```
R' = 0
for i = N−1 to 0
   R = {R' << 1, $A_i$}
   D = R − B
   if D < 0 then    $Q_i$ = 0, R' = R    // R < B
   else             $Q_i$ = 1, R' = D    // R ≥ B
R = R'
```

The *partial remainder* R is initialized to 0. The most significant bit of the dividend A then becomes the least significant bit of R. The divisor B is repeatedly subtracted from this partial remainder to determine whether it fits. If the difference D is negative (i.e., the sign bit of D is 1), then

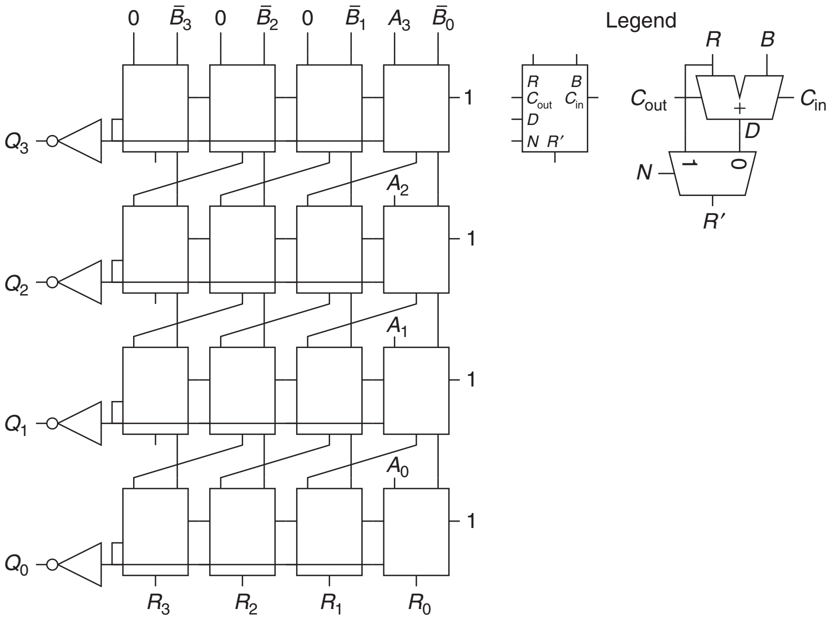


Figure 5.20 **Array divider**

the quotient bit Q_i is 0 and the difference is discarded. Otherwise, Q_i is 1, and the partial remainder is updated to be the difference. In any event, the partial remainder is then doubled (left-shifted by one column), the next most significant bit of A becomes the least significant bit of R, and the process repeats. The result satisfies $\frac{A}{B} = Q + \frac{R}{B}$.

Figure 5.20 shows a schematic of a 4-bit array divider. The divider computes A/B and produces a quotient Q and a remainder R. The legend shows the symbol and schematic for each block in the array divider. The signal N indicates whether $R - B$ is negative. It is obtained from the D output of the leftmost block in the row, which is the sign of the difference.

The delay of an N-bit array divider increases proportionally to N^2 because the carry must ripple through all N stages in a row before the sign is determined and the multiplexer selects R or D. This repeats for all N rows. Division is a slow and expensive operation in hardware and therefore should be used as infrequently as possible.

5.2.8 Further Reading

Computer arithmetic could be the subject of an entire text. *Digital Arithmetic*, by Ercegovac and Lang, is an excellent overview of the entire field. *CMOS VLSI Design*, by Weste and Harris, covers high-performance circuit designs for arithmetic operations.

5.3 NUMBER SYSTEMS

Computers operate on both integers and fractions. So far, we have only considered representing signed or unsigned integers, as introduced in Section 1.4. This section introduces fixed- and floating-point number systems that can represent rational numbers. Fixed-point numbers are analogous to decimals; some of the bits represent the integer part, and the rest represent the fraction. Floating-point numbers are analogous to scientific notation, with a mantissa and an exponent.

5.3.1 Fixed-Point Number Systems

Fixed-point notation has an implied *binary point* between the integer and fraction bits, analogous to the decimal point between the integer and fraction digits of an ordinary decimal number. For example, Figure 5.21(a) shows a fixed-point number with four integer bits and four fraction bits. Figure 5.21(b) shows the implied binary point in blue, and Figure 5.21(c) shows the equivalent decimal value.

Signed fixed-point numbers can use either two's complement or sign/magnitude notation. Figure 5.22 shows the fixed-point representation of −2.375 using both notations with four integer and four fraction bits. The implicit binary point is shown in blue for clarity. In sign/magnitude form, the most significant bit is used to indicate the sign. The two's complement representation is formed by inverting the bits of the absolute value and adding a 1 to the least significant (rightmost) bit. In this case, the least significant bit position is in the 2^{-4} column.

Like all binary number representations, fixed-point numbers are just a collection of bits. There is no way of knowing the existence of the binary point except through agreement of those people interpreting the number.

(a) 01101100

(b) 0110.1100

(c) $2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

Figure 5.21 Fixed-point notation of 6.75 with four integer bits and four fraction bits

(a) 0010.0110

(b) 1010.0110

(c) 1101.1010

Figure 5.22 Fixed-point representation of −2.375: (a) absolute value, (b) sign and magnitude, (c) two's complement

---

Example 5.4 ARITHMETIC WITH FIXED-POINT NUMBERS

Compute 0.75 + −0.625 using fixed-point numbers.

Solution: First convert 0.625, the magnitude of the second number, to fixed-point binary notation. $0.625 \geq 2^{-1}$, so there is a 1 in the 2^{-1} column, leaving $0.625 - 0.5 = 0.125$. Because $0.125 < 2^{-2}$, there is a 0 in the 2^{-2} column. Because $0.125 \geq 2^{-3}$, there is a 1 in the 2^{-3} column, leaving $0.125 - 0.125 = 0$. Thus, there must be a 0 in the 2^{-4} column. Putting this all together, $0.625_{10} = 0000.1010_2$.

Use two's complement representation for signed numbers so that addition works correctly. Figure 5.23 shows the conversion of −0.625 to fixed-point two's complement notation.

Figure 5.24 shows the fixed-point binary addition and the decimal equivalent for comparison. Note that the leading 1 in the binary fixed-point addition of Figure 5.24(a) is discarded from the 8-bit result.

Fixed-point number systems are commonly used for banking and financial applications that require precision but not a large range.

0000.1010	Binary Magnitude
1111.0101	One's Complement
+ 1	Add 1
1111.0110	Two's Complement

Figure 5.23 **Fixed-point two's complement conversion**

$$\begin{array}{r} 0000.1100 \\ +\ 1111.0110 \\ \hline 10000.0010 \end{array} \qquad \begin{array}{r} 0.75 \\ +\ (-0.625) \\ \hline 0.125 \end{array}$$

(a) (b)

Figure 5.24 **Addition: (a) binary fixed-point, (b) decimal equivalent**

5.3.2 Floating-Point Number Systems*

Floating-point numbers are analogous to scientific notation. They circumvent the limitation of having a constant number of integer and fractional bits, allowing the representation of very large and very small numbers. Like scientific notation, floating-point numbers have a *sign*, *mantissa* (M), *base* (B), and *exponent* (E), as shown in Figure 5.25. For example, the number 4.1×10^3 is the decimal scientific notation for 4100. It has a mantissa of 4.1, a base of 10, and an exponent of 3. The decimal point *floats* to the position right after the most significant digit. Floating-point numbers are base 2 with a binary mantissa. 32 bits are used to represent 1 sign bit, 8 exponent bits, and 23 mantissa bits.

$$\pm M \times B^E$$

Figure 5.25 **Floating-point numbers**

Example 5.5 32-BIT FLOATING-POINT NUMBERS

Show the floating-point representation of the decimal number 228.

Solution: First convert the decimal number into binary: $228_{10} = 11100100_2 = 1.11001_2 \times 2^7$. Figure 5.26 shows the 32-bit encoding, which will be modified later for efficiency. The sign bit is positive (0), the 8 exponent bits give the value 7, and the remaining 23 bits are the mantissa.

1 bit	8 bits	23 bits
0	00000111	111 0010 0000 0000 0000 0000
Sign	**Exponent**	**Mantissa**

Figure 5.26 **32-bit floating-point version 1**

In binary floating-point, the first bit of the mantissa (to the left of the binary point) is always 1 and therefore need not be stored. It is called the *implicit leading one*. Figure 5.27 shows the modified floating-point representation of $228_{10} = 11100100_2 \times 2^0 = 1.11001_2 \times 2^7$. The implicit leading one is not included in the 23-bit mantissa for efficiency. Only the fraction bits are stored. This frees up an extra bit for useful data.

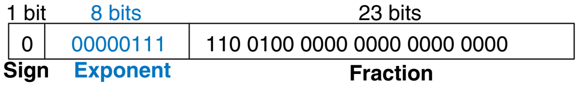


Figure 5.27 Floating-point version 2

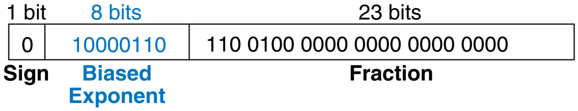


Figure 5.28 IEEE 754 floating-point notation

We make one final modification to the exponent field. The exponent needs to represent both positive and negative exponents. To do so, floating-point uses a *biased* exponent, which is the original exponent plus a constant bias. 32-bit floating-point uses a bias of 127. For example, for the exponent 7, the biased exponent is $7 + 127 = 134 = 10000110_2$. For the exponent −4, the biased exponent is: $-4 + 127 = 123 = 01111011_2$. Figure 5.28 shows $1.11001_2 \times 2^7$ represented in floating-point notation with an implicit leading one and a biased exponent of 134 (7 + 127). This notation conforms to the IEEE 754 floating-point standard.

As may be apparent, there are many reasonable ways to represent floating-point numbers. For many years, computer manufacturers used incompatible floating-point formats. Results from one computer could not directly be interpreted by another computer.

The Institute of Electrical and Electronics Engineers solved this problem by creating the *IEEE 754 floating-point standard* in 1985 defining floating-point numbers. This floating-point format is now almost universally used and is the one discussed in this section.

Special Cases: 0, ±∞, and NaN

The IEEE floating-point standard has special cases to represent numbers such as zero, infinity, and illegal results. For example, representing the number zero is problematic in floating-point notation because of the implicit leading one. Special codes with exponents of all 0's or all l's are reserved for these special cases. Table 5.2 shows the floating-point representations of 0, ±∞, and NaN. As with sign/magnitude numbers, floating-point has both positive and negative 0. NaN is used for numbers that don't exist, such as $\sqrt{-1}$ or $\log_2(-5)$.

Single- and Double-Precision Formats

So far, we have examined 32-bit floating-point numbers. This format is also called *single-precision*, *single*, or *float*. The IEEE 754 standard also

Table 5.2 IEEE 754 floating-point notations for 0, ±∞, and NaN

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

Table 5.3 Single- and double-precision floating-point formats

Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits
single	32	1	8	23
double	64	1	11	52

defines 64-bit *double-precision* numbers (also called *doubles*) that provide greater precision and greater range. Table 5.3 shows the number of bits used for the fields in each format.

Excluding the special cases mentioned earlier, normal single-precision numbers span a range of $\pm 1.175494 \times 10^{-38}$ to $\pm 3.402824 \times 10^{38}$. They have a precision of about seven significant decimal digits (because $2^{-24} \approx 10^{-7}$). Similarly, normal double-precision numbers span a range of $\pm 2.22507385850720 \times 10^{-308}$ to $\pm 1.79769313486232 \times 10^{308}$ and have a precision of about 15 significant decimal digits.

Floating-point cannot represent some numbers exactly, like 1.7. However, when you type 1.7 into your calculator, you see exactly 1.7, not 1.69999. . . . To handle this, some applications, such as calculators and financial software, use *binary coded decimal* (*BCD*) numbers or formats with a base 10 exponent. BCD numbers encode each decimal digit using four bits with a range of 0 to 9. For example, the BCD fixed-point notation of 1.7 with four integer bits and four fraction bits would be 0001.0111. Of course, nothing is free. The cost is increased complexity in arithmetic hardware and wasted encodings (A–F encodings are not used), and thus decreased performance. So for compute-intensive applications, floating-point is much faster.

Rounding

Arithmetic results that fall outside of the available precision must round to a neighboring number. The rounding modes are: round down, round up, round toward zero, and round to nearest. The default rounding mode is round to nearest. In the round to nearest mode, if two numbers are equally near, the one with a 0 in the least significant position of the fraction is chosen.

Recall that a number *overflows* when its magnitude is too large to be represented. Likewise, a number *underflows* when it is too tiny to be represented. In round to nearest mode, overflows are rounded up to $\pm\infty$ and underflows are rounded down to 0.

Floating-Point Addition

Addition with floating-point numbers is not as simple as addition with two's complement numbers. The steps for adding floating-point numbers with the same sign are as follows:

1. Extract exponent and fraction bits.
2. Prepend leading 1 to form the mantissa.
3. Compare exponents.
4. Shift smaller mantissa if necessary.
5. Add mantissas.
6. Normalize mantissa and adjust exponent if necessary.
7. Round result.
8. Assemble exponent and fraction back into floating-point number.

Figure 5.29 shows the floating-point addition of 7.875 (1.11111×2^2) and 0.1875 (1.1×2^{-3}). The result is 8.0625 (1.0000001×2^3). After the fraction and exponent bits are extracted and the implicit leading 1 is prepended in steps 1 and 2, the exponents are compared by subtracting the smaller exponent from the larger exponent. The result is the number of bits by which the smaller number is shifted to the right to align the implied binary point (i.e., to make the exponents equal) in step 4. The aligned numbers are added. Because the sum has a mantissa that is greater than or equal to 2.0, the result is normalized by shifting it to the right one bit and incrementing the exponent. In this example, the result is exact, so no rounding is necessary. The result is stored in floating-point notation by removing the implicit leading one of the mantissa and prepending the sign bit.

Floating-point arithmetic is usually done in hardware to make it fast. This hardware, called the *floating-point unit* (*FPU*), is typically distinct from the *central processing unit* (*CPU*). The infamous *floating-point division* (*FDIV*) bug in the Pentium FPU cost Intel $475 million to recall and replace defective chips. The bug occurred simply because a lookup table was not loaded correctly.

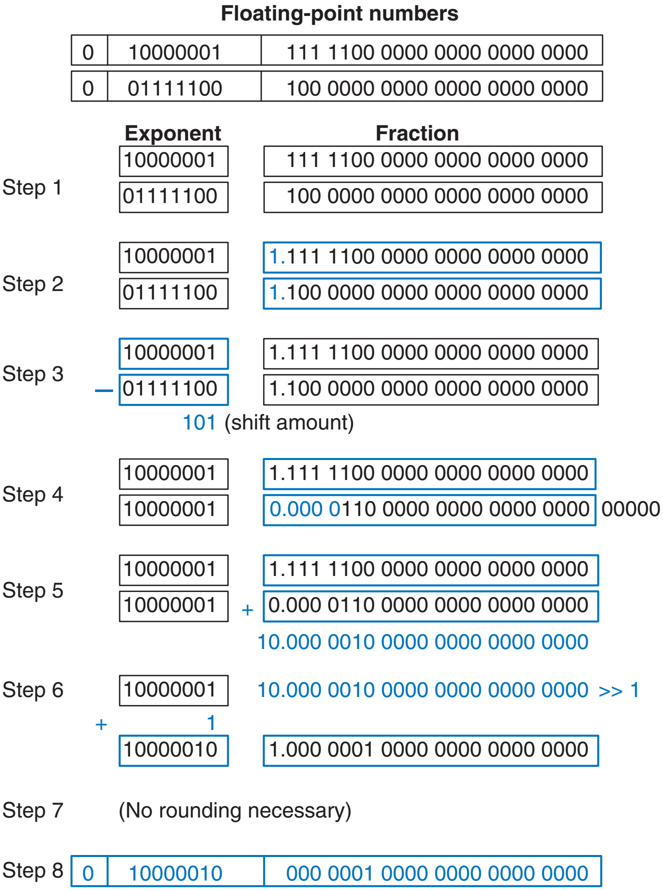

Floating-point numbers
0 | 10000001 | 111 1100 0000 0000 0000 0000
0 | 01111100 | 100 0000 0000 0000 0000 0000
Exponent
Fraction
Step 1
10000001 | 111 1100 0000 0000 0000 0000
01111100 | 100 0000 0000 0000 0000 0000
Step 2
10000001 | 1.111 1100 0000 0000 0000 0000
01111100 | 1.100 0000 0000 0000 0000 0000
Step 3
10000001 | 1.111 1100 0000 0000 0000 0000
– 01111100 | 1.100 0000 0000 0000 0000 0000
101 (shift amount)
Step 4
10000001 | 1.111 1100 0000 0000 0000 0000
10000001 | 0.000 0110 0000 0000 0000 0000 00000
Step 5
10000001 | 1.111 1100 0000 0000 0000 0000
10000001 + 0.000 0110 0000 0000 0000 0000
10.000 0010 0000 0000 0000 0000
Step 6
10000001 | 10.000 0010 0000 0000 0000 0000 >> 1
+ 1
10000010 | 1.000 0001 0000 0000 0000 0000
Step 7
(No rounding necessary)
Step 8
0 | 10000010 | 000 0001 0000 0000 0000 0000

Figure 5.29 **Floating-point addition**

5.4 SEQUENTIAL BUILDING BLOCKS

This section examines sequential building blocks, including counters and shift registers.

5.4.1 Counters

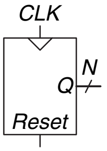


Figure 5.30 **Counter symbol**

An N-bit *binary counter*, shown in Figure 5.30, is a sequential arithmetic circuit with clock and reset inputs and an N-bit output Q. *Reset* initializes the output to 0. The counter then advances through all 2^N possible outputs in binary order, incrementing on the rising edge of the clock.

Figure 5.31 shows an N-bit counter composed of an adder and a resettable register. On each cycle, the counter adds 1 to the value stored in the register. HDL Example 5.5 describes a binary counter with asynchronous reset.

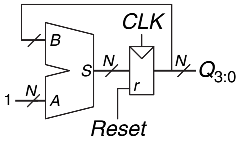


Figure 5.31 ***N*-bit counter**

Other types of counters, such as Up/Down counters, are explored in Exercises 5.43 through 5.46.

HDL Example 5.5 COUNTER

SystemVerilog

```
module counter #(parameter N = 8)
               (input  logic clk,
                input  logic reset,
                output logic [N-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= q + 1;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity counter is
  generic(N: integer := 8);
  port(clk, reset: in  STD_LOGIC;
       q:          out STD_LOGIC_VECTOR(N-1 downto 0));
end;

architecture synth of counter is
begin
  process(clk, reset) begin
    if reset then              q <= (OTHERS => '0');
    elsif rising_edge(clk) then q <= q + '1';
    end if;
  end process;
end;
```

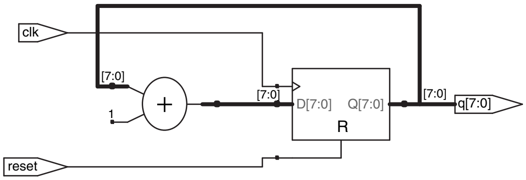


Figure 5.32 **Synthesized counter**

5.4.2 Shift Registers

A *shift register* has a clock, a serial input S_{in}, a serial output S_{out}, and N parallel outputs $Q_{N-1:0}$, as shown in Figure 5.33. On each rising edge of the clock, a new bit is shifted in from S_{in} and all the subsequent contents are shifted forward. The last bit in the shift register is available at S_{out}. Shift registers can be viewed as *serial-to-parallel converters*. The input is provided serially (one bit at a time) at S_{in}. After N cycles, the past N inputs are available in parallel at Q.

A shift register can be constructed from N flip-flops connected in series, as shown in Figure 5.34. Some shift registers also have a reset signal to initialize all of the flip-flops.

A related circuit is a *parallel-to-serial* converter that loads N bits in parallel, then shifts them out one at a time. A shift register can be modified to perform both serial-to-parallel and parallel-to-serial operations by adding a parallel input $D_{N-1:0}$, and a control signal *Load*, as shown in Figure 5.35. When *Load* is asserted, the flip-flops are loaded in parallel from the D inputs. Otherwise, the shift register shifts normally. HDL Example 5.6 describes such a shift register.

Don't confuse *shift registers* with the *shifters* from Section 5.2.5. Shift registers are sequential logic blocks that shift in a new bit on each clock edge. Shifters are unclocked combinational logic blocks that shift an input by a specified amount.

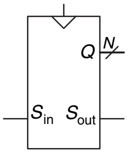


Figure 5.33 Shift register symbol

Scan Chains*

Shift registers are often used to test sequential circuits using a technique called *scan chains*. Testing combinational circuits is relatively straightforward. Known inputs called *test vectors* are applied, and the outputs are checked against the expected result. Testing sequential circuits is more difficult, because the circuits have state. Starting from a known initial condition, a large number of cycles of test vectors may be needed to put the circuit into a desired state. For example, testing that the most significant bit of a 32-bit counter advances from 0 to 1 requires resetting the counter, then applying 2^{31} (about two billion) clock pulses!

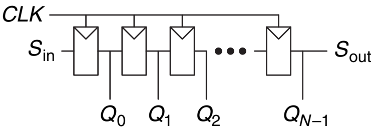


Figure 5.34 Shift register schematic

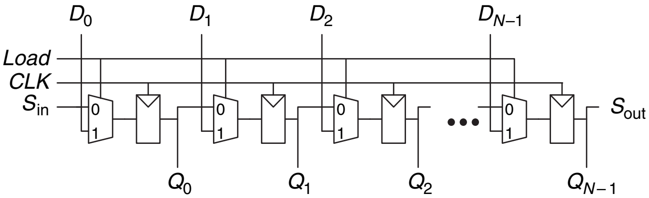


Figure 5.35 Shift register with parallel load

HDL Example 5.6 SHIFT REGISTER WITH PARALLEL LOAD

SystemVerilog

```
module shiftreg #(parameter N = 8)
                 (input  logic        clk,
                  input  logic        reset, load,
                  input  logic        sin,
                  input  logic [N-1:0] d,
                  output logic [N-1:0] q,
                  output logic        sout);

  always_ff @(posedge clk, posedge reset)
    if (reset)      q <= 0;
    else if (load)  q <= d;
    else            q <= {q[N-2:0], sin};

  assign sout = q[N-1];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;

entity shiftreg is
  generic(N: integer := 8);
  port(clk, reset: in  STD_LOGIC;
       load, sin:  in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(N-1 downto 0);
       q:          out STD_LOGIC_VECTOR(N-1 downto 0);
       sout:       out STD_LOGIC);
end;

architecture synth of shiftreg is
begin
  process(clk, reset) begin
    if reset = '1' then   q <= (OTHERS => '0');
    elsif rising_edge(clk) then
      if load then        q <= d;
      else                q <= q(N-2 downto 0) & sin;
      end if;
    end if;
  end process;

  sout <= q(N-1);
end;
```

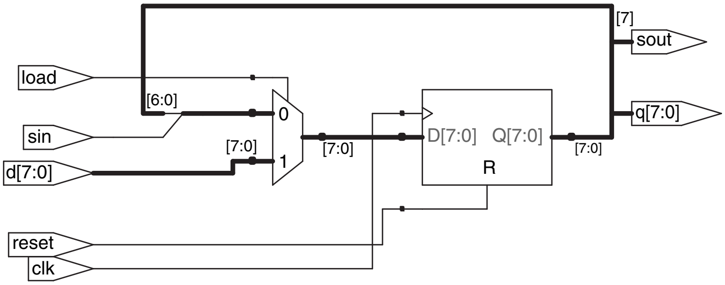


Figure 5.36 Synthesized shiftreg

To solve this problem, designers like to be able to directly observe and control all the state of the machine. This is done by adding a test mode in which the contents of all flip-flops can be read out or loaded with desired values. Most systems have too many flip-flops to dedicate individual pins to read and write each flip-flop. Instead, all the flip-flops in the system are connected together into a shift register called a scan chain. In normal operation, the flip-flops load data from their *D* input and ignore the scan chain. In test mode, the flip-flops serially shift their

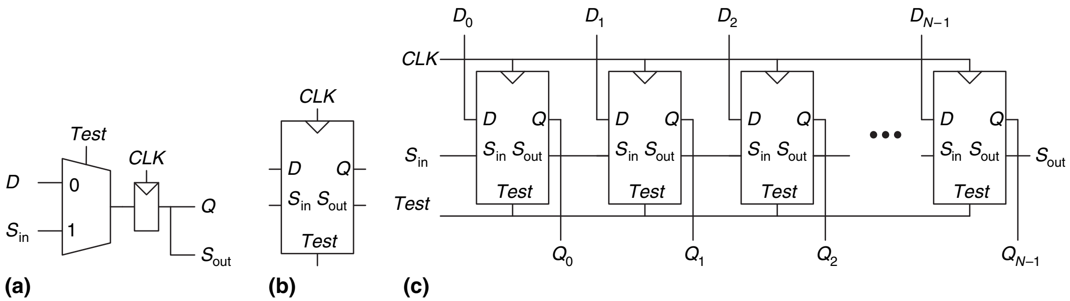


Figure 5.37 **Scannable flip-flop: (a) schematic, (b) symbol, and (c) *N*-bit scannable register**

contents out and shift in new contents using S_{in} and S_{out}. The load multiplexer is usually integrated into the flip-flop to produce a *scannable flip-flop*. Figure 5.37 shows the schematic and symbol for a scannable flip-flop and illustrates how the flops are cascaded to build an *N*-bit scannable register.

For example, the 32-bit counter could be tested by shifting in the pattern 011111...111 in test mode, counting for one cycle in normal mode, then shifting out the result, which should be 100000...000. This requires only $32 + 1 + 32 = 65$ cycles.

5.5 MEMORY ARRAYS

The previous sections introduced arithmetic and sequential circuits for manipulating data. Digital systems also require *memories* to store the data used and generated by such circuits. Registers built from flip-flops are a kind of memory that stores small amounts of data. This section describes *memory arrays* that can efficiently store large amounts of data.

The section begins with an overview describing characteristics shared by all memory arrays. It then introduces three types of memory arrays: dynamic random access memory (DRAM), static random access memory (SRAM), and read only memory (ROM). Each memory differs in the way it stores data. The section briefly discusses area and delay trade-offs and shows how memory arrays are used, not only to store data but also to perform logic functions. The section finishes with the HDL for a memory array.

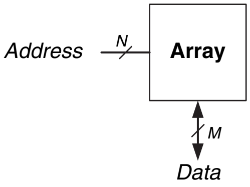


Figure 5.38 **Generic memory array symbol**

5.5.1 Overview

Figure 5.38 shows a generic symbol for a memory array. The memory is organized as a two-dimensional array of memory cells. The memory reads or writes the contents of one of the rows of the array. This row is specified by an *Address*. The value read or written is called *Data*. An array with *N*-bit addresses and *M*-bit data has 2^N rows and *M* columns. Each row of data is called a *word*. Thus, the array contains 2^N *M*-bit words.

Figure 5.39 shows a memory array with two address bits and three data bits. The two address bits specify one of the four rows (data words) in the array. Each data word is three bits wide. Figure 5.39(b) shows some possible contents of the memory array.

The *depth* of an array is the number of rows, and the *width* is the number of columns, also called the word size. The size of an array is given as *depth* × *width*. Figure 5.39 is a 4-word × 3-bit array, or simply 4 × 3 array. The symbol for a 1024-word × 32-bit array is shown in Figure 5.40. The total size of this array is 32 kilobits (Kb).

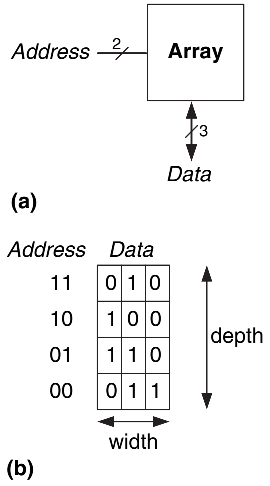


Figure 5.39 4 × 3 memory array: (a) symbol, (b) function

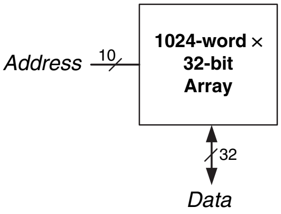


Figure 5.40 32 Kb array: depth = 2^{10} = 1024 words, width = 32 bits

Bit Cells

Memory arrays are built as an array of *bit cells*, each of which stores 1 bit of data. Figure 5.41 shows that each bit cell is connected to a *wordline* and a *bitline*. For each combination of address bits, the memory asserts a single wordline that activates the bit cells in that row. When the wordline is HIGH, the stored bit transfers to or from the bitline. Otherwise, the bitline is disconnected from the bit cell. The circuitry to store the bit varies with memory type.

To read a bit cell, the bitline is initially left floating (Z). Then the wordline is turned ON, allowing the stored value to drive the bitline to 0 or 1. To write a bit cell, the bitline is strongly driven to the desired value. Then the wordline is turned ON, connecting the bitline to the stored bit. The strongly driven bitline overpowers the contents of the bit cell, writing the desired value into the stored bit.

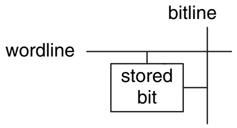


Figure 5.41 Bit cell

Organization

Figure 5.42 shows the internal organization of a 4 × 3 memory array. Of course, practical memories are much larger, but the behavior of larger arrays can be extrapolated from the smaller array. In this example, the array stores the data from Figure 5.39(b).

During a memory read, a wordline is asserted, and the corresponding row of bit cells drives the bitlines HIGH or LOW. During a memory write, the bitlines are driven HIGH or LOW first and then a wordline is asserted, allowing the bitline values to be stored in that row of bit cells. For example, to read *Address* 10, the bitlines are left floating, the decoder asserts $wordline_2$, and the data stored in that row of bit cells (100) reads out onto the *Data* bitlines. To write the value 001 to *Address* 11, the bitlines are driven to the value 001, then $wordline_3$ is asserted and the new value (001) is stored in the bit cells.

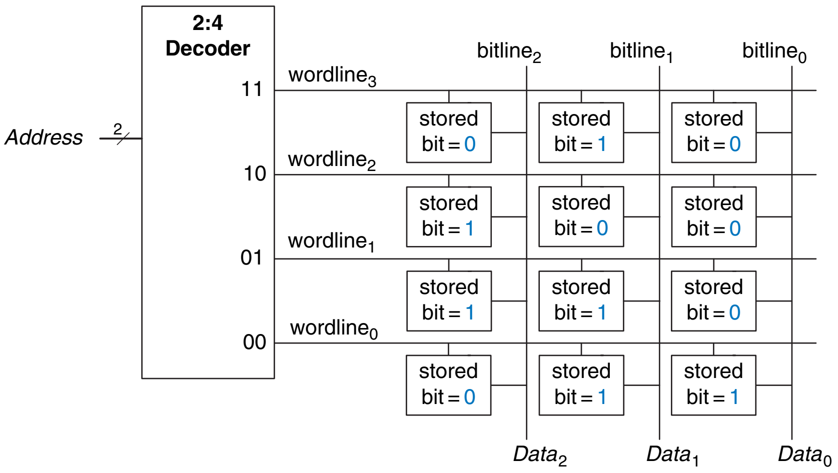

2:4
Decoder
Address
2
11
10
01
00
wordline$_3$
wordline$_2$
wordline$_1$
wordline$_0$
bitline$_2$
bitline$_1$
bitline$_0$
stored bit = 0
stored bit = 1
stored bit = 0
stored bit = 1
stored bit = 0
stored bit = 0
stored bit = 1
stored bit = 1
stored bit = 0
stored bit = 0
stored bit = 1
stored bit = 1
Data$_2$
Data$_1$
Data$_0$

Figure 5.42 4 × 3 memory array

Memory Ports

All memories have one or more *ports*. Each port gives read and/or write access to one memory address. The previous examples were all single-ported memories.

Multiported memories can access several addresses simultaneously. Figure 5.43 shows a three-ported memory with two read ports and one write port. Port 1 reads the data from address *A1* onto the read data output *RD1*. Port 2 reads the data from address *A2* onto *RD2*. Port 3 writes the data from the write data input *WD3* into address *A3* on the rising edge of the clock if the write enable *WE3* is asserted.

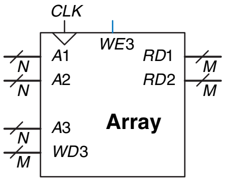

CLK
WE3
A1
A2
A3
WD3
RD1
RD2
Array
N
N
N
M
M
M

Figure 5.43 Three-ported memory

Memory Types

Memory arrays are specified by their size (depth × width) and the number and type of ports. All memory arrays store data as an array of bit cells, but they differ in how they store bits.

Memories are classified based on how they store bits in the bit cell. The broadest classification is *random access memory* (*RAM*) versus *read only memory* (*ROM*). RAM is *volatile*, meaning that it loses its data when the power is turned off. ROM is *nonvolatile*, meaning that it retains its data indefinitely, even without a power source.

RAM and ROM received their names for historical reasons that are no longer very meaningful. RAM is called *random* access memory because any data word is accessed with the same delay as any other. In contrast, a sequential access memory, such as a tape recorder, accesses nearby data more quickly than faraway data (e.g., at the other end of the tape). ROM is called *read only* memory because, historically, it could only be read but not written. These names are confusing, because ROMs are randomly accessed too. Worse yet, most modern ROMs can be written as well as read! The important distinction to remember is that RAMs are volatile and ROMs are nonvolatile.

The two major types of RAMs are *dynamic RAM* (*DRAM*) and *static RAM* (*SRAM*). Dynamic RAM stores data as a charge on a capacitor, whereas static RAM stores data using a pair of cross-coupled inverters. There are many flavors of ROMs that vary by how they are written and erased. These various types of memories are discussed in the subsequent sections.

Robert Dennard, 1932–. Invented DRAM in 1966 at IBM. Although many were skeptical that the idea would work, by the mid-1970s DRAM was in virtually all computers. He claims to have done little creative work until, arriving at IBM, they handed him a patent notebook and said, "put all your ideas in there." Since 1965, he has received 35 patents in semiconductors and microelectronics. (Photo courtesy of IBM.)

5.5.2 Dynamic Random Access Memory (DRAM)

Dynamic RAM (*DRAM*, pronounced "dee-ram") stores a bit as the presence or absence of charge on a capacitor. Figure 5.44 shows a DRAM bit cell. The bit value is stored on a capacitor. The nMOS transistor behaves as a switch that either connects or disconnects the capacitor from the bitline. When the wordline is asserted, the nMOS transistor turns ON, and the stored bit value transfers to or from the bitline.

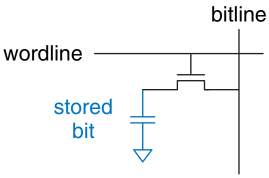


Figure 5.44 **DRAM bit cell**

As shown in Figure 5.45(a), when the capacitor is charged to V_{DD}, the stored bit is 1; when it is discharged to GND (Figure 5.45(b)), the stored bit is 0. The capacitor node is *dynamic* because it is not actively driven HIGH or LOW by a transistor tied to V_{DD} or GND.

Upon a read, data values are transferred from the capacitor to the bitline. Upon a write, data values are transferred from the bitline to the capacitor. Reading destroys the bit value stored on the capacitor, so the data word must be restored (rewritten) after each read. Even when DRAM is not read, the contents must be refreshed (read and rewritten) every few milliseconds, because the charge on the capacitor gradually leaks away.

5.5.3 Static Random Access Memory (SRAM)

Static RAM (*SRAM*, pronounced "es-ram") is *static* because stored bits do not need to be refreshed. Figure 5.46 shows an SRAM bit cell. The data bit is stored on cross-coupled inverters like those described in Section 3.2. Each cell has two outputs, bitline and $\overline{\text{bitline}}$. When the wordline is asserted, both nMOS transistors turn on, and data values are transferred to or from the bitlines. Unlike DRAM, if noise degrades the value of the stored bit, the cross-coupled inverters restore the value.

5.5.4 Area and Delay

Flip-flops, SRAMs, and DRAMs are all volatile memories, but each has different area and delay characteristics. Table 5.4 shows a comparison of these three types of volatile memory. The data bit stored in a flip-flop is available

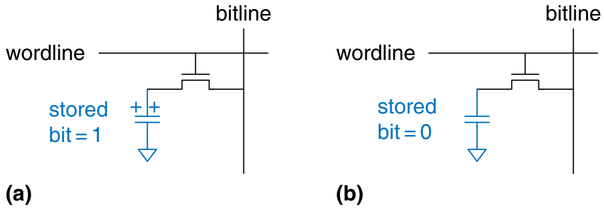


Figure 5.45 DRAM stored values

Table 5.4 Memory comparison

Memory Type	Transistors per Bit Cell	Latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow

immediately at its output. But flip-flops take at least 20 transistors to build. Generally, the more transistors a device has, the more area, power, and cost it requires. DRAM latency is longer than that of SRAM because its bitline is not actively driven by a transistor. DRAM must wait for charge to move (relatively) slowly from the capacitor to the bitline. DRAM also fundamentally has lower throughput than SRAM, because it must refresh data periodically and after a read. DRAM technologies such as *synchronous DRAM* (*SDRAM*) and *double data rate* (*DDR*) SDRAM have been developed to overcome this problem. SDRAM uses a clock to pipeline memory accesses. DDR SDRAM, sometimes called simply DDR, uses both the rising and falling edges of the clock to access data, thus doubling the throughput for a given clock speed. DDR was first standardized in 2000 and ran at 100 to 200 MHz. Later standards, DDR2, DDR3, and DDR4, increased the clock speeds, with speeds in 2012 being over 1 GHz.

Memory latency and throughput also depend on memory size; larger memories tend to be slower than smaller ones if all else is the same. The best memory type for a particular design depends on the speed, cost, and power constraints.

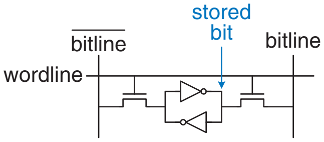


Figure 5.46 SRAM bit cell

5.5.5 Register Files

Digital systems often use a number of registers to store temporary variables. This group of registers, called a *register file*, is usually built as a small, multiported SRAM array, because it is more compact than an array of flip-flops.

Figure 5.47 shows a 32-register × 32-bit three-ported register file built from a three-ported memory similar to that of Figure 5.43. The register

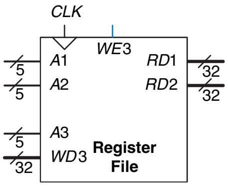


Figure 5.47 32 × 32 register file with two read ports and one write port

file has two read ports (*A1*/*RD1* and *A2*/*RD2*) and one write port (*A3*/*WD3*). The 5-bit addresses, *A1*, *A2*, and *A3*, can each access all $2^5 = 32$ registers. So, two registers can be read and one register written simultaneously.

5.5.6 Read Only Memory

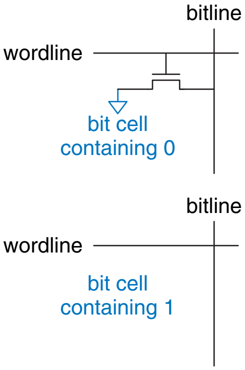


Figure 5.48 **ROM bit cells containing 0 and 1**

Read only memory (*ROM*) stores a bit as the presence or absence of a transistor. Figure 5.48 shows a simple ROM bit cell. To read the cell, the bitline is weakly pulled HIGH. Then the wordline is turned ON. If the transistor is present, it pulls the bitline LOW. If it is absent, the bitline remains HIGH. Note that the ROM bit cell is a combinational circuit and has no state to "forget" if power is turned off.

The contents of a ROM can be indicated using *dot notation*. Figure 5.49 shows the dot notation for a 4-word × 3-bit ROM containing the data from Figure 5.39. A dot at the intersection of a row (wordline) and a column (bitline) indicates that the data bit is 1. For example, the top wordline has a single dot on $Data_1$, so the data word stored at *Address* 11 is 010.

Conceptually, ROMs can be built using two-level logic with a group of AND gates followed by a group of OR gates. The AND gates produce all possible minterms and hence form a decoder. Figure 5.50 shows the ROM of Figure 5.49 built using a decoder and OR gates. Each dotted row in Figure 5.49 is an input to an OR gate in Figure 5.50. For data bits with a single dot, in this case $Data_0$, no OR gate is needed. This representation of a ROM is interesting because it shows how the ROM can perform any two-level logic function. In practice, ROMs are built from transistors instead of logic gates to reduce their size and cost. Section 5.6.3 explores the transistor-level implementation further.

The contents of the ROM bit cell in Figure 5.48 are specified during manufacturing by the presence or absence of a transistor in each bit cell. A *programmable ROM* (*PROM*, pronounced like the dance) places a

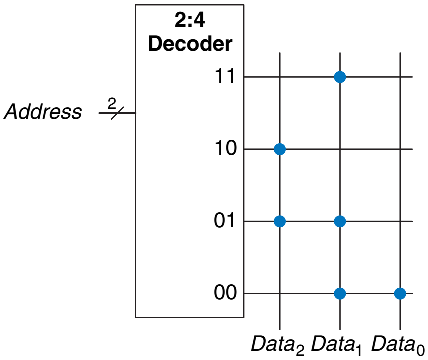


Figure 5.49 **4 × 3 ROM: dot notation**

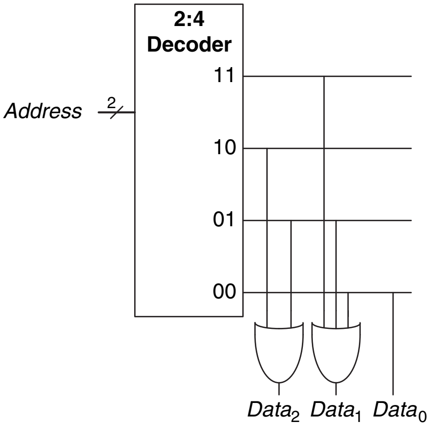


Figure 5.50 **4 × 3 ROM implementation using gates**

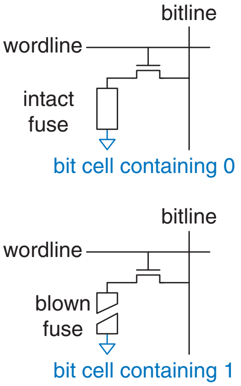


Figure 5.51 **Fuse-programmable ROM bit cell**

transistor in every bit cell but provides a way to connect or disconnect the transistor to ground.

Figure 5.51 shows the bit cell for a *fuse-programmable ROM*. The user programs the ROM by applying a high voltage to selectively blow fuses. If the fuse is present, the transistor is connected to GND and the cell holds a 0. If the fuse is destroyed, the transistor is disconnected from ground and the cell holds a 1. This is also called a one-time programmable ROM, because the fuse cannot be repaired once it is blown.

Reprogrammable ROMs provide a reversible mechanism for connecting or disconnecting the transistor to GND. *Erasable PROMs* (*EPROMs*, pronounced "e-proms") replace the nMOS transistor and fuse with a *floating-gate transistor*. The floating gate is not physically attached to any other wires. When suitable high voltages are applied, electrons tunnel through an insulator onto the floating gate, turning on the transistor and connecting the bitline to the wordline (decoder output). When the EPROM is exposed to intense ultraviolet (UV) light for about half an hour, the electrons are knocked off the floating gate, turning the transistor off. These actions are called *programming* and *erasing*, respectively. *Electrically erasable PROMs* (*EEPROMs*, pronounced "e-e-proms" or "double-e proms") and *Flash* memory use similar principles but include circuitry on the chip for erasing as well as programming, so no UV light is necessary. EEPROM bit cells are individually erasable; Flash memory erases larger blocks of bits and is cheaper because fewer erasing circuits are needed. In 2012, Flash memory cost about $1 per GB, and the price continues to drop by 30 to 40% per year. Flash has become an extremely popular way to store large

Fujio Masuoka, 1944–. Received a Ph.D. in electrical engineering from Tohoku University, Japan. Developed memories and high-speed circuits at Toshiba from 1971 to 1994. Invented Flash memory as an unauthorized project pursued during nights and weekends in the late 1970s. Flash received its name because the process of erasing the memory reminds one of the flash of a camera. Toshiba was slow to commercialize the idea; Intel was first to market in 1988. Flash has grown into a $25 billion per year market. Dr. Masuoka later joined the faculty at Tohoku University and is working to develop a 3-dimensional transistor.

amounts of data in portable battery-powered systems such as cameras and music players.

In summary, modern ROMs are not really read only; they can be programmed (written) as well. The difference between RAM and ROM is that ROMs take a longer time to write but are nonvolatile.

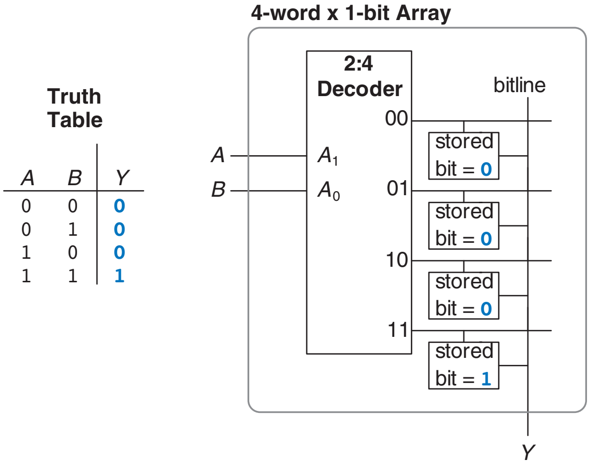
5.5.7 Logic Using Memory Arrays

Although they are used primarily for data storage, memory arrays can also perform combinational logic functions. For example, the $Data_2$ output of the ROM in Figure 5.49 is the XOR of the two *Address* inputs. Likewise $Data_0$ is the NAND of the two inputs. A 2^N-word × M-bit memory can perform any combinational function of N inputs and M outputs. For example, the ROM in Figure 5.49 performs three functions of two inputs.

Memory arrays used to perform logic are called *lookup tables* (*LUTs*). Figure 5.52 shows a 4-word × 1-bit memory array used as a lookup table to perform the function $Y = AB$. Using memory to perform logic, the user can look up the output value for a given input combination (address). Each address corresponds to a row in the truth table, and each data bit corresponds to an output value.

5.5.8 Memory HDL

HDL Example 5.7 describes a 2^N-word × M-bit RAM. The RAM has a synchronous enabled write. In other words, writes occur on the rising

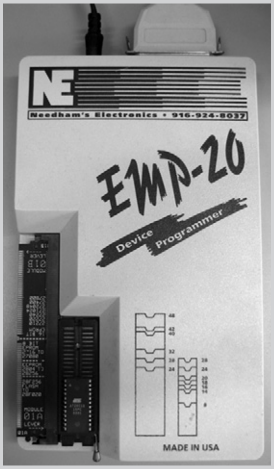


A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 5.52 4-word × 1-bit memory array used as a lookup table

Flash memory drives with Universal Serial Bus (USB) connectors have replaced floppy disks and CDs for sharing files because Flash costs have dropped so dramatically.

Programmable ROMs can be configured with a device programmer like the one shown below. The device programmer is attached to a computer, which specifies the type of ROM and the data values to program. The device programmer blows fuses or injects charge onto a floating gate on the ROM. Thus the programming process is sometimes called *burning* a ROM.



HDL Example 5.7 RAM

SystemVerilog

```
module ram #(parameter N = 6, M = 32)
            (input  logic         clk,
             input  logic         we,
             input  logic [N-1:0] adr,
             input  logic [M-1:0] din,
             output logic [M-1:0] dout);

  logic [M-1:0] mem [2**N-1:0];

  always_ff @(posedge clk)
    if (we) mem [adr] <= din;

  assign dout = mem[adr];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD_UNSIGNED.ALL;

entity ram_array is
  generic(N: integer := 6; M: integer := 32);
  port(clk,
       we:   in  STD_LOGIC;
       adr:  in  STD_LOGIC_VECTOR(N-1 downto 0);
       din:  in  STD_LOGIC_VECTOR(M-1 downto 0);
       dout: out STD_LOGIC_VECTOR(M-1 downto 0));
end;

architecture synth of ram_array is
  type mem_array is array ((2**N-1) downto 0)
       of STD_LOGIC_VECTOR (M-1 downto 0);
  signal mem: mem_array;
begin
  process(clk) begin
    if rising_edge(clk) then
      if we then mem(TO_INTEGER(adr)) <= din;
      end if;
    end if;
  end process;

  dout <= mem(TO_INTEGER(adr));
end;
```

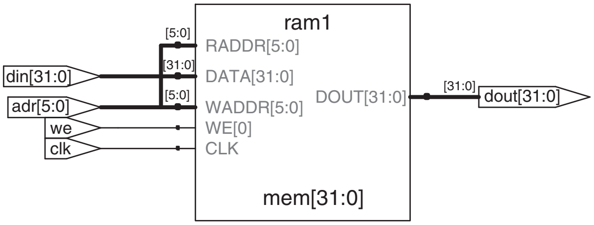


Figure 5.53 **Synthesized ram**

edge of the clock if the write enable *we* is asserted. Reads occur immediately. When power is first applied, the contents of the RAM are unpredictable.

HDL Example 5.8 describes a 4-word × 3-bit ROM. The contents of the ROM are specified in the HDL `case` statement. A ROM as small as this one may be synthesized into logic gates rather than an array. Note that the seven-segment decoder from HDL Example 4.24 synthesizes into a ROM in Figure 4.20.

HDL Example 5.8 ROM

SystemVerilog

```
module rom(input  logic [1:0] adr,
           output logic [2:0] dout):

  always_comb
    case(adr)
      2'b00: dout <= 3'b011;
      2'b01: dout <= 3'b110;
      2'b10: dout <= 3'b100;
      2'b11: dout <= 3'b010;
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity rom is
  port(adr:  in  STD_LOGIC_VECTOR(1 downto 0);
       dout: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of rom is
begin
  process(all) begin
    case adr is
      when "00" => dout <= "011";
      when "01" => dout <= "110";
      when "10" => dout <= "100";
      when "11" => dout <= "010";
    end case;
  end process;
end;
```

5.6 LOGIC ARRAYS

Like memory, gates can be organized into regular arrays. If the connections are made programmable, these *logic arrays* can be configured to perform any function without the user having to connect wires in specific ways. The regular structure simplifies design. Logic arrays are mass produced in large quantities, so they are inexpensive. Software tools allow users to map logic designs onto these arrays. Most logic arrays are also reconfigurable, allowing designs to be modified without replacing the hardware. Reconfigurability is valuable during development and is also useful in the field, because a system can be upgraded by simply downloading the new configuration.

This section introduces two types of logic arrays: programmable logic arrays (PLAs), and field programmable gate arrays (FPGAs). PLAs, the older technology, perform only combinational logic functions. FPGAs can perform both combinational and sequential logic.

5.6.1 Programmable Logic Array

Programmable logic arrays (*PLAs*) implement two-level combinational logic in sum-of-products (SOP) form. PLAs are built from an AND array followed by an OR array, as shown in Figure 5.54. The inputs (in true

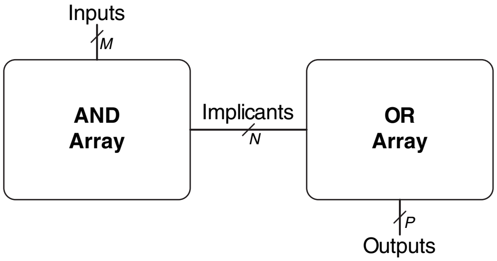


Figure 5.54 $M \times N \times P$-bit PLA

and complementary form) drive an AND array, which produces implicants, which in turn are ORed together to form the outputs. An $M \times N \times P$-bit PLA has M inputs, N implicants, and P outputs.

Figure 5.55 shows the dot notation for a $3 \times 3 \times 2$-bit PLA performing the functions $X = \overline{A}\,\overline{B}C + AB\overline{C}$ and $Y = A\overline{B}$. Each row in the AND array forms an implicant. Dots in each row of the AND array indicate which literals comprise the implicant. The AND array in Figure 5.55 forms three implicants: $\overline{A}\,\overline{B}C$, $AB\overline{C}$, and $A\overline{B}$. Dots in the OR array indicate which implicants are part of the output function.

Figure 5.56 shows how PLAs can be built using two-level logic. An alternative implementation is given in Section 5.6.3.

ROMs can be viewed as a special case of PLAs. A 2^M-word $\times$ N-bit ROM is simply an $M \times 2^M \times N$-bit PLA. The decoder behaves as an AND plane that produces all 2^M minterms. The ROM array behaves as an OR plane that produces the outputs. If the function does not

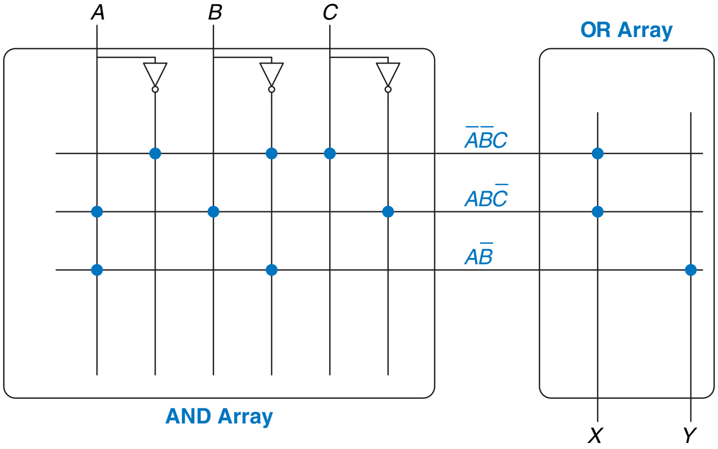


Figure 5.55 $3 \times 3 \times 2$-bit PLA: dot notation

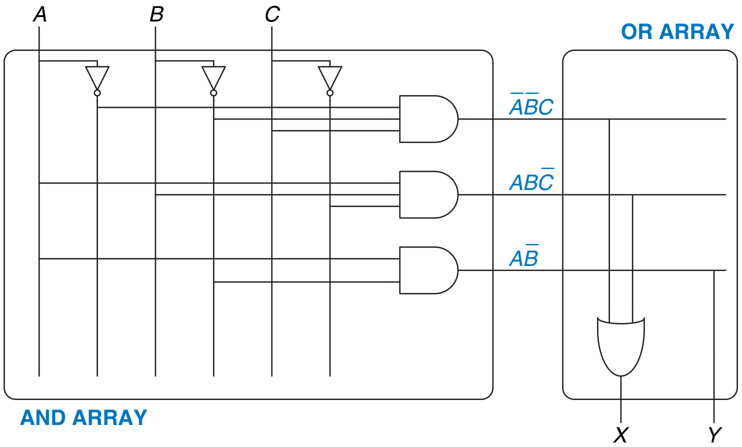


Figure 5.56 **3 × 3 × 2-bit PLA using two-level logic**

FPGAs are the brains of many consumer products, including automobiles, medical equipment, and media devices like MP3 players. The Mercedes Benz S-Class series, for example, has over a dozen Xilinx FPGAs or PLDs for uses ranging from entertainment to navigation to cruise control systems. FPGAs allow for quick time to market and make debugging or adding features late in the design process easier.

depend on all 2^M minterms, a PLA is likely to be smaller than a ROM. For example, an 8-word × 2-bit ROM is required to perform the same functions performed by the 3 × 3 × 2-bit PLA shown in Figures 5.55 and 5.56.

Simple programmable logic devices (*SPLDs*) are souped-up PLAs that add registers and various other features to the basic AND/OR planes. However, SPLDs and PLAs have largely been displaced by FPGAs, which are more flexible and efficient for building large systems.

5.6.2 Field Programmable Gate Array

A *field programmable gate array* (*FPGA*) is an array of reconfigurable gates. Using software programming tools, a user can implement designs on the FPGA using either an HDL or a schematic. FPGAs are more powerful and more flexible than PLAs for several reasons. They can implement both combinational and sequential logic. They can also implement multi-level logic functions, whereas PLAs can only implement two-level logic. Modern FPGAs integrate other useful features such as built-in multipliers, high-speed I/Os, data converters including analog-to-digital converters, large RAM arrays, and processors.

FPGAs are built as an array of configurable *logic elements* (*LEs*), also referred to as *configurable logic blocks* (*CLBs*). Each LE can be configured to perform combinational or sequential functions. Figure 5.57 shows a general block diagram of an FPGA. The LEs are surrounded by *input/output elements* (*IOEs*) for interfacing with the outside world. The IOEs connect LE inputs and outputs to pins on the chip package. LEs can connect to other LEs and IOEs through programmable routing channels.

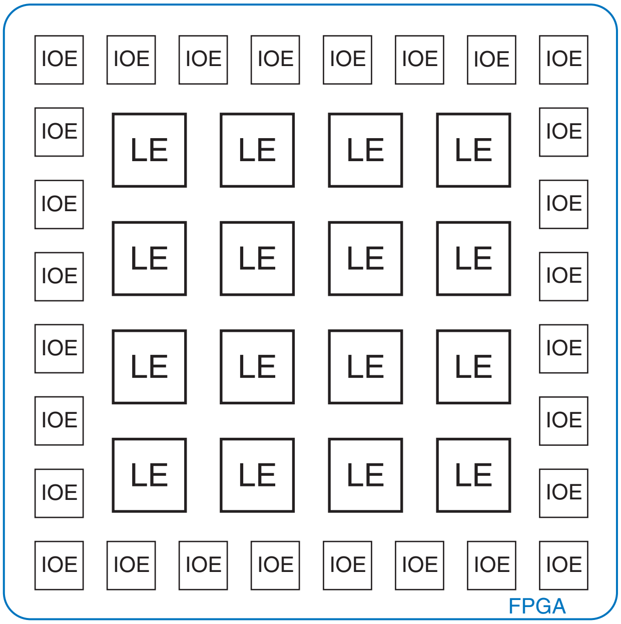


Figure 5.57 **General FPGA layout**

Two of the leading FPGA manufacturers are Altera Corp. and Xilinx, Inc. Figure 5.58 shows a single LE from Altera's Cyclone IV FPGA introduced in 2009. The key elements of the LE are a 4-input lookup table (LUT) and a 1-bit register. The LE also contains configurable multiplexers to route signals through the LE. The FPGA is configured by specifying the contents of the lookup tables and the select signals for the multiplexers.

The Cyclone IV LE has one 4-input LUT and one flip-flop. By loading the appropriate values into the lookup table, the LUT can be configured to perform any function of up to four variables. Configuring the FPGA also involves choosing the select signals that determine how the multiplexers route data through the LE and to neighboring LEs and IOEs. For example, depending on the multiplexer configuration, the LUT may receive one of its inputs from either *data 3* or the output of the LE's own register. The other three inputs always come from *data 1*, *data 2*, and *data 4*. The *data 1-4* inputs come from IOEs or the outputs of other LEs, depending on routing external to the LE. The LUT output either goes directly to the LE output for combinational functions, or it can be fed through the flip-flop for registered functions. The flip-flop input comes from its own LUT output, the *data 3* input, or the register output of the previous LE. Additional hardware includes support for addition using

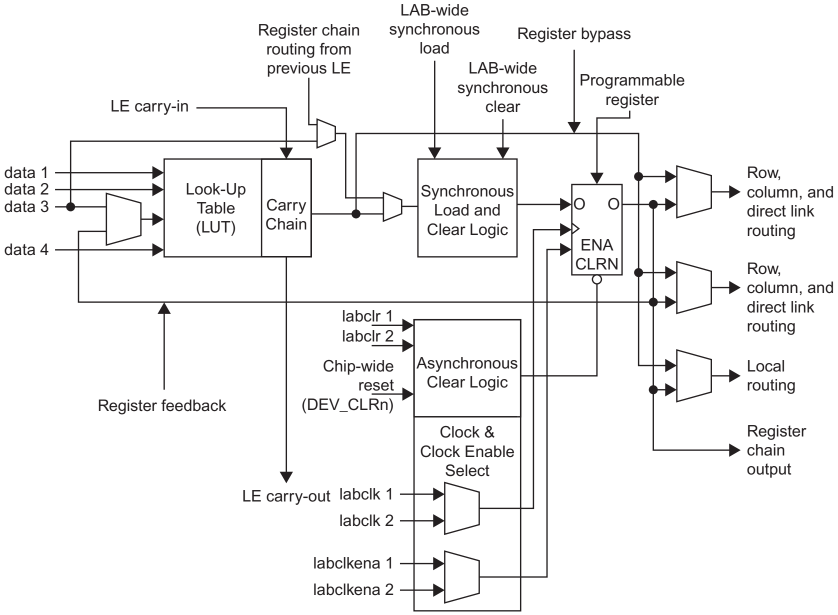

Register chain routing from previous LE
LAB-wide synchronous load
LAB-wide synchronous clear
Register bypass
Programmable register
LE carry-in
data 1
data 2
data 3
data 4
Look-Up Table (LUT)
Carry Chain
Synchronous Load and Clear Logic
ENA
CLRN
Row, column, and direct link routing
Row, column, and direct link routing
Local routing
Register chain output
labclr 1
labclr 2
Chip-wide reset (DEV_CLRn)
Asynchronous Clear Logic
Clock & Clock Enable Select
Register feedback
LE carry-out
labclk 1
labclk 2
labclkena 1
labclkena 2

Figure 5.58 **Cyclone IV Logic Element (LE)**

(Reproduced with permission from the Altera Cyclone™ IV Handbook © 2010 Altera Corporation.)

the carry chain hardware, other multiplexers for routing, and flip-flop enable and reset. Altera groups 16 LEs together to create a *logic array block* (*LAB*) and provides local connections between LEs within the LAB.

In summary, the Cyclone IV LE can perform one combinational and/or registered function which can involve up to four variables. Other brands of FPGAs are organized somewhat differently, but the same general principles apply. For example, Xilinx's 7-series FPGAs use 6-input LUTs instead of 4-input LUTs.

The designer configures an FPGA by first creating a schematic or HDL description of the design. The design is then synthesized onto the FPGA. The synthesis tool determines how the LUTs, multiplexers, and routing channels should be configured to perform the specified functions. This configuration information is then downloaded to the FPGA. Because Cyclone IV FPGAs store their configuration information in SRAM, they are easily reprogrammed. The FPGA may download its SRAM contents from a computer in the laboratory or from an EEPROM chip when the system is turned on. Some manufacturers include an EEPROM directly on the FPGA or use one-time programmable fuses to configure the FPGA.

Example 5.6 FUNCTIONS BUILT USING LEs

Explain how to configure one or more Cyclone IV LEs to perform the following functions: (a) $X = \overline{A}\,\overline{B}C + AB\overline{C}$ and $Y = A\overline{B}$ (b) $Y = JKLMPQR$; (c) a divide-by-3 counter with binary state encoding (see Figure 3.29(a)). You may show interconnection between LEs as needed.

Solution: (a) Configure two LEs. One LUT computes X and the other LUT computes Y, as shown in Figure 5.59. For the first LE, inputs *data 1*, *data 2*, and *data 3* are A, B, and C, respectively (these connections are set by the routing channels). *data 4* is a don't care but must be tied to something, so it is tied to 0. For the second LE, inputs *data 1* and *data 2* are A and B; the other LUT inputs are don't cares and are tied to 0. Configure the final multiplexers to select the combinational outputs from the LUTs to produce X and Y. In general, a single LE can compute any function of up to four input variables in this fashion.

(b) Configure the LUT of the first LE to compute $X = JKLM$ and the LUT on the second LE to compute $Y = XPQR$. Configure the final multiplexers to select the combinational outputs X and Y from each LE. This configuration is shown in Figure 5.60. Routing channels between LEs, indicated by the dashed blue lines, connect the output of LE 1 to the input of LE 2. In general, a group of LEs can compute functions of N input variables in this manner.

(c) The FSM has two bits of state ($S_{1:0}$) and one output (Y). The next state depends on the two bits of current state. Use two LEs to compute the next state from the current state, as shown in Figure 5.61. Use the two flip-flops, one from each LE, to hold this state. The flip-flops have a reset input that can be connected to an external *Reset* signal. The registered outputs are fed back to the LUT inputs

(*A*) data 1	(*B*) data 2	(*C*) data 3	data 4	(*X*) LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0

(*A*) data 1	(*B*) data 2	data 3	data 4	(*Y*) LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0

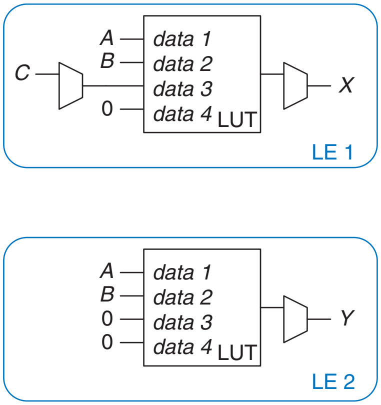


Figure 5.59 LE configuration for two functions of up to four inputs each

Figure 5.60 **LE configuration for one function of more than four inputs**

(*J*) *data 1*	(*K*) *data 2*	(*L*) *data 3*	(*M*) *data 4*	(*X*) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(*P*) *data 1*	(*Q*) *data 2*	(*R*) *data 3*	(*X*) *data 4*	(*Y*) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

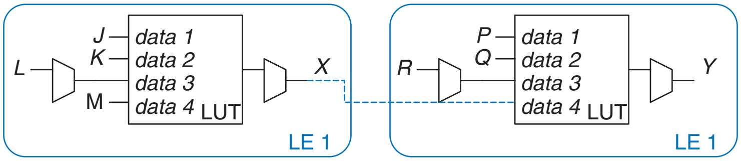
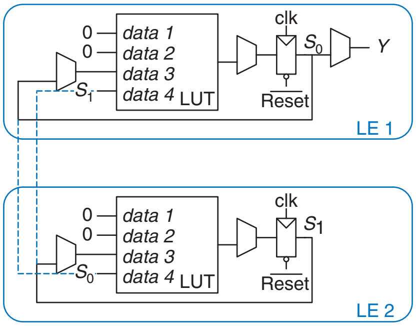

J
K
L
M
data 1
data 2
data 3
data 4
LUT
X
LE 1
P
Q
R
data 1
data 2
data 3
data 4
LUT
Y
LE 1

Figure 5.61 **LE configuration for FSM with two bits of state**

data 1	*data 2*	(S_0) *data 3*	(S_1) *data 4*	(S_0') LUT output
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	0

data 1	*data 2*	(S_1) *data 3*	(S_0) *data 4*	(S_1') LUT output
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	0


0
0
data 1
data 2
data 3
data 4
LUT
S1
clk
S0
Reset
Y
LE 1
0
0
data 1
data 2
data 3
data 4
LUT
S0
clk
S1
Reset
LE 2

using the multiplexer on *data 3* and routing channels between LEs, as indicated by the dashed blue lines. In general, another LE might be necessary to compute the output *Y*. However, in this case $Y = S_0$, so *Y* can come from LE 1. Hence, the entire FSM fits in two LEs. In general, an FSM requires at least one LE for each bit of state, and it may require more LEs for the output or next state logic if they are too complex to fit in a single LUT.

Example 5.7 LE DELAY

Alyssa P. Hacker is building a finite state machine that must run at 200 MHz. She uses a Cyclone IV GX FPGA with the following specifications: $t_{LE} = 381$ ps per LE, $t_{\text{setup}} = 76$ ps, and $t_{pcq} = 199$ ps for all flip-flops. The wiring delay between LEs is 246 ps. Assume the hold time for the flip-flops is 0. What is the maximum number of LEs her design can use?

Solution: Alyssa uses Equation 3.13 to solve for the maximum propagation delay of the logic: $t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}})$.

Thus, $t_{pd} = 5$ ns – (0.199 ns + 0.076 ns), so $t_{pd} \leq 4.725$ ns. The delay of each LE plus wiring delay between LEs, $t_{LE+wire}$, is 381 ps + 246 ps = 627 ps. The maximum number of LEs, N, is $Nt_{LE+wire} \leq 4.725$ ns. Thus, $N = 7$.

5.6.3 Array Implementations*

To minimize their size and cost, ROMs and PLAs commonly use pseudo-nMOS or dynamic circuits (see Section 1.7.8) instead of conventional logic gates.

Figure 5.62(a) shows the dot notation for a 4 × 3-bit ROM that performs the following functions: $X = A \oplus B$, $Y = \overline{A} + B$, and $Z = \overline{A}\,\overline{B}$. These are the same functions as those of Figure 5.49, with the address inputs renamed A and B and the data outputs renamed X, Y, and Z. The pseudo-nMOS implementation is given in Figure 5.62(b). Each decoder output is connected to the gates of the nMOS transistors in its row. Remember that in pseudo-nMOS circuits, the weak pMOS transistor pulls the output HIGH *only if* there is no path to GND through the pulldown (nMOS) network.

Pull-down transistors are placed at every junction without a dot. The dots from the dot notation diagram of Figure 5.62(a) are left visible in Figure 5.62(b) for easy comparison. The weak pull-up transistors pull the

Many ROMs and PLAs use dynamic circuits in place of pseudo-nMOS circuits. Dynamic gates turn the pMOS transistor ON for only part of the time, saving power when the pMOS is OFF and the result is not needed. Aside from this, dynamic and pseudo-nMOS memory arrays are similar in design and behavior.

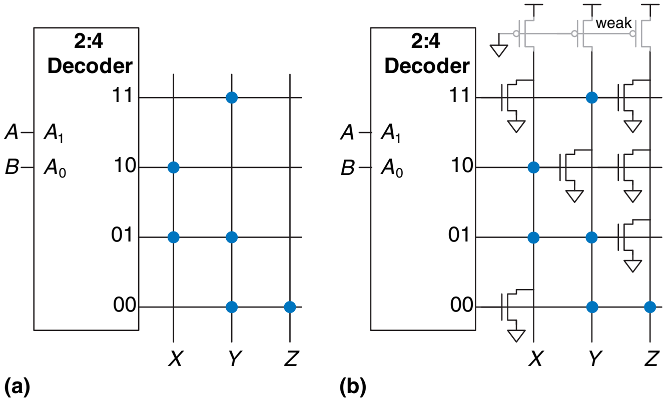


Figure 5.62 ROM implementation: (a) dot notation, (b) pseudo-nMOS circuit

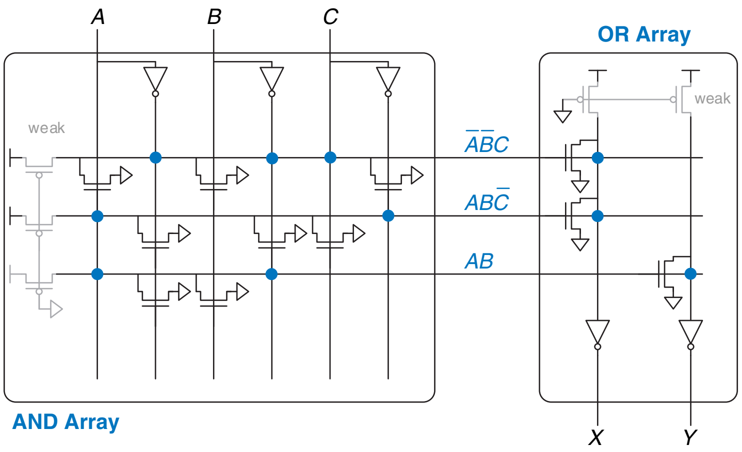


Figure 5.63 **3 × 3 × 2-bit PLA using pseudo-nMOS circuits**

output HIGH for each wordline without a pull-down transistor. For example, when $AB = 11$, the 11 wordline is HIGH and transistors on X and Z turn on and pull those outputs LOW. The Y output has no transistor connecting to the 11 wordline, so Y is pulled HIGH by the weak pull-up.

PLAs can also be built using pseudo-nMOS circuits, as shown in Figure 5.63 for the PLA from Figure 5.55. Pull-down (nMOS) transistors are placed on the *complement* of dotted literals in the AND array and on dotted rows in the OR array. The columns in the OR array are sent through an inverter before they are fed to the output bits. Again, the blue dots from the dot notation diagram of Figure 5.55 are left visible in Figure 5.63 for easy comparison.

5.7 SUMMARY

This chapter introduced digital building blocks used in many digital systems. These blocks include arithmetic circuits such as adders, subtractors, comparators, shifters, multipliers, and dividers; sequential circuits such as counters and shift registers; and arrays for memory and logic. The chapter also explored fixed-point and floating-point representations of fractional numbers. In Chapter 7, we use these building blocks to build a microprocessor.

Adders form the basis of most arithmetic circuits. A half adder adds two 1-bit inputs, A and B, and produces a sum and a carry out. A full adder extends the half adder to also accept a carry in. N full adders can be cascaded to form a carry propagate adder (CPA) that adds two N-bit numbers. This type of CPA is called a ripple-carry adder because the carry ripples through each of the full adders. Faster CPAs can be constructed using lookahead or prefix techniques.

A subtractor negates the second input and adds it to the first. A magnitude comparator subtracts one number from another and determines the relative value based on the sign of the result. A multiplier forms partial products using AND gates, then sums these bits using full adders. A divider repeatedly subtracts the divisor from the partial remainder and checks the sign of the difference to determine the quotient bits. A counter uses an adder and a register to increment a running count.

Fractional numbers are represented using fixed-point or floating-point forms. Fixed-point numbers are analogous to decimals, and floating-point numbers are analogous to scientific notation. Fixed-point numbers use ordinary arithmetic circuits, whereas floating-point numbers require more elaborate hardware to extract and process the sign, exponent, and mantissa.

Large memories are organized into arrays of words. The memories have one or more ports to read and/or write the words. Volatile memories, such as SRAM and DRAM, lose their state when the power is turned off. SRAM is faster than DRAM but requires more transistors. A register file is a small multiported SRAM array. Nonvolatile memories, called ROMs, retain their state indefinitely. Despite their names, most modern ROMs can be written.

Arrays are also a regular way to build logic. Memory arrays can be used as lookup tables to perform combinational functions. PLAs are composed of dedicated connections between configurable AND and OR arrays; they only implement combinational logic. FPGAs are composed of many small lookup tables and registers; they implement combinational and sequential logic. The lookup table contents and their interconnections can be configured to perform any logic function. Modern FPGAs are easy to reprogram and are large and cheap enough to build highly sophisticated digital systems, so they are widely used in low- and medium-volume commercial products as well as in education.

Exercises

Exercise 5.1 What is the delay for the following types of 64-bit adders? Assume that each two-input gate delay is 150 ps and that a full adder delay is 450 ps.

(a) a ripple-carry adder

(b) a carry-lookahead adder with 4-bit blocks

(c) a prefix adder

Exercise 5.2 Design two adders: a 64-bit ripple-carry adder and a 64-bit carry-lookahead adder with 4-bit blocks. Use only two-input gates. Each two-input gate is 15 μm^2, has a 50 ps delay, and has 20 fF of total gate capacitance. You may assume that the static power is negligible.

(a) Compare the area, delay, and power of the adders (operating at 100 MHz and 1.2 V).

(b) Discuss the trade-offs between power, area, and delay.

Exercise 5.3 Explain why a designer might choose to use a ripple-carry adder instead of a carry-lookahead adder.

Exercise 5.4 Design the 16-bit prefix adder of Figure 5.7 in an HDL. Simulate and test your module to prove that it functions correctly.

Exercise 5.5 The prefix network shown in Figure 5.7 uses black cells to compute all of the prefixes. Some of the block propagate signals are not actually necessary. Design a "gray cell" that receives G and P signals for bits $i{:}k$ and $k-1{:}j$ but produces only $G_{i:j}$, not $P_{i:j}$. Redraw the prefix network, replacing black cells with gray cells wherever possible.

Exercise 5.6 The prefix network shown in Figure 5.7 is not the only way to calculate all of the prefixes in logarithmic time. The *Kogge-Stone* network is another common prefix network that performs the same function using a different connection of black cells. Research Kogge-Stone adders and draw a schematic similar to Figure 5.7 showing the connection of black cells in a Kogge-Stone adder.

Exercise 5.7 Recall that an N-input priority encoder has $\log_2 N$ outputs that encodes which of the N inputs gets priority (see Exercise 2.36).

(a) Design an N-input priority encoder that has delay that increases logarithmically with N. Sketch your design and give the delay of the circuit in terms of the delay of its circuit elements.

(b) Code your design in an HDL. Simulate and test your module to prove that it functions correctly.

Exercise 5.8 Design the following comparators for 32-bit numbers. Sketch the schematics.

(a) not equal

(b) greater than

(c) less than or equal to

Exercise 5.9 Design the 32-bit ALU shown in Figure 5.15 using your favorite HDL. You can make the top-level module either behavioral or structural.

Exercise 5.10 Add an *Overflow* output to the 32-bit ALU from Exercise 5.9. The output is TRUE when the result of the adder overflows. Otherwise, it is FALSE.

(a) Write a Boolean equation for the *Overflow* output.

(b) Sketch the Overflow circuit.

(c) Design the modified ALU in an HDL.

Exercise 5.11 Add a *Zero* output to the 32-bit ALU from Exercise 5.9. The output is TRUE when $Y == 0$.

Exercise 5.12 Write a testbench to test the 32-bit ALU from Exercise 5.9, 5.10, or 5.11. Then use it to test the ALU. Include any test vector files necessary. Be sure to test enough corner cases to convince a reasonable skeptic that the ALU functions correctly.

Exercise 5.13 Design a shifter that always shifts a 32-bit input left by 2 bits. The input and output are both 32 bits. Explain the design in words and sketch a schematic. Implement your design in your favorite HDL.

Exercise 5.14 Design 4-bit left and right rotators. Sketch a schematic of your design. Implement your design in your favorite HDL.

Exercise 5.15 Design an 8-bit left shifter using only 24 2:1 multiplexers. The shifter accepts an 8-bit input A and a 3-bit shift amount, $shamt_{2:0}$. It produces an 8-bit output Y. Sketch the schematic.

Exercise 5.16 Explain how to build any N-bit shifter or rotator using only $N\log_2 N$ 2:1 multiplexers.

Exercise 5.17 The *funnel shifter* in Figure 5.64 can perform any N-bit shift or rotate operation. It shifts a $2N$-bit input right by k bits. The output Y is the N least significant bits of the result. The most significant N bits of the input are called B and the least significant N bits are called C. By choosing appropriate values of B, C, and k, the funnel shifter can perform any type of shift or rotate. Explain what these values should be in terms of A, *shamt*, and N for

(a) logical right shift of A by *shamt*

(b) arithmetic right shift of A by *shamt*

(c) left shift of A by *shamt*

(d) right rotate of A by *shamt*

(e) left rotate of A by *shamt*

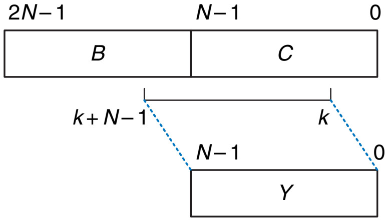


Figure 5.64 **Funnel shifter**

Exercise 5.18 Find the critical path for the 4 × 4 multiplier from Figure 5.18 in terms of an AND gate delay (t_{AND}) and an adder delay (t_{FA}) What is the delay of an $N \times N$ multiplier built in the same way?

Exercise 5.19 Find the critical path for the 4 × 4 divider from Figure 5.20 in terms of a 2:1 mux delay (t_{MUX}), an adder delay (t_{FA}), and an inverter delay (t_{INV}). What is the delay of an $N \times N$ divider built in the same way?

Exercise 5.20 Design a multiplier that handles two's complement numbers.

Exercise 5.21 A *sign extension unit* extends a two's complement number from M to N ($N > M$) bits by copying the most significant bit of the input into the upper bits of the output (see Section 1.4.6). It receives an M-bit input A and produces an N-bit output Y. Sketch a circuit for a sign extension unit with a 4-bit input and an 8-bit output. Write the HDL for your design.

Exercise 5.22 A *zero extension unit* extends an unsigned number from M to N bits ($N > M$) by putting zeros in the upper bits of the output. Sketch a circuit for a zero extension unit with a 4-bit input and an 8-bit output. Write the HDL for your design.

Exercise 5.23 Compute $111001.000_2/001100.000_2$ in binary using the standard division algorithm from elementary school. Show your work.

Exercise 5.24 What is the range of numbers that can be represented by the following number systems?

(a) 24-bit unsigned fixed-point numbers with 12 integer bits and 12 fraction bits

(b) 24-bit sign and magnitude fixed-point numbers with 12 integer bits and 12 fraction bits

(c) 24-bit two's complement fixed-point numbers with 12 integer bits and 12 fraction bits

Exercise 5.25 Express the following base 10 numbers in 16-bit fixed-point sign/magnitude format with eight integer bits and eight fraction bits. Express your answer in hexadecimal.

(a) −13.5625

(b) 42.3125

(c) −17.15625

Exercise 5.26 Express the following base 10 numbers in 12-bit fixed-point sign/magnitude format with six integer bits and six fraction bits. Express your answer in hexadecimal.

(a) −30.5

(b) 16.25

(c) −8.078125

Exercise 5.27 Express the base 10 numbers in Exercise 5.25 in 16-bit fixed-point two's complement format with eight integer bits and eight fraction bits. Express your answer in hexadecimal.

Exercise 5.28 Express the base 10 numbers in Exercise 5.26 in 12-bit fixed-point two's complement format with six integer bits and six fraction bits. Express your answer in hexadecimal.

Exercise 5.29 Express the base 10 numbers in Exercise 5.25 in IEEE 754 single-precision floating-point format. Express your answer in hexadecimal.

Exercise 5.30 Express the base 10 numbers in Exercise 5.26 in IEEE 754 single-precision floating-point format. Express your answer in hexadecimal.

Exercise 5.31 Convert the following two's complement binary fixed-point numbers to base 10. The implied binary point is explicitly shown to aid in your interpretation.

(a) 0101.1000

(b) 1111.1111

(c) 1000.0000

Exercise 5.32 Repeat Exercise 5.31 for the following two's complement binary fixed-point numbers.

(a) 011101.10101

(b) 100110.11010

(c) 101000.00100

Exercise 5.33 When adding two floating-point numbers, the number with the smaller exponent is shifted. Why is this? Explain in words and give an example to justify your explanation.

Exercise 5.34 Add the following IEEE 754 single-precision floating-point numbers.

(a) C0123456 + 81C564B7

(b) D0B10301 + D1B43203

(c) 5EF10324 + 5E039020

Exercise 5.35 Add the following IEEE 754 single-precision floating-point numbers.

(a) C0D20004 + 72407020

(b) C0D20004 + 40DC0004

(c) (5FBE4000 + 3FF80000) + DFDE4000
(Why is the result counterintuitive? Explain.)

Exercise 5.36 Expand the steps in section 5.3.2 for performing floating-point addition to work for negative as well as positive floating-point numbers.

Exercise 5.37 Consider IEEE 754 single-precision floating-point numbers.

(a) How many numbers can be represented by IEEE 754 single-precision floating-point format? You need not count ±∞ or NaN.

(b) How many additional numbers could be represented if ±∞ and NaN were not represented?

(c) Explain why ±∞ and NaN are given special representations.

Exercise 5.38 Consider the following decimal numbers: 245 and 0.0625.

(a) Write the two numbers using single-precision floating-point notation. Give your answers in hexadecimal.

(b) Perform a magnitude comparison of the two 32-bit numbers from part (a). In other words, interpret the two 32-bit numbers as two's complement numbers and compare them. Does the integer comparison give the correct result?

(c) You decide to come up with a new single-precision floating-point notation. Everything is the same as the IEEE 754 single-precision floating-point standard, except that you represent the exponent using two's complement instead of a bias. Write the two numbers using your new standard. Give your answers in hexadecimal.

(e) Does integer comparison work with your new floating-point notation from part (d)?

(f) Why is it convenient for integer comparison to work with floating-point numbers?

Exercise 5.39 Design a single-precision floating-point adder using your favorite HDL. Before coding the design in an HDL, sketch a schematic of your design. Simulate and test your adder to prove to a skeptic that it functions correctly. You may consider positive numbers only and use round toward zero (truncate). You may also ignore the special cases given in Table 5.2.

Exercise 5.40 In this problem, you will explore the design of a 32-bit floating-point multiplier. The multiplier has two 32-bit floating-point inputs and produces a 32-bit floating-point output. You may consider positive numbers only and use round toward zero (truncate). You may also ignore the special cases given in Table 5.2.

(a) Write the steps necessary to perform 32-bit floating-point multiplication.

(b) Sketch the schematic of a 32-bit floating-point multiplier.

(c) Design a 32-bit floating-point multiplier in an HDL. Simulate and test your multiplier to prove to a skeptic that it functions correctly.

Exercise 5.41 In this problem, you will explore the design of a 32-bit prefix adder.

(a) Sketch a schematic of your design.

(b) Design the 32-bit prefix adder in an HDL. Simulate and test your adder to prove that it functions correctly.

(c) What is the delay of your 32-bit prefix adder from part (a)? Assume that each two-input gate delay is 100 ps.

(d) Design a pipelined version of the 32-bit prefix adder. Sketch the schematic of your design. How fast can your pipelined prefix adder run? You may assume a sequencing overhead ($t_{pcq} + t_{\text{setup}}$) of 80 ps. Make the design run as fast as possible.

(e) Design the pipelined 32-bit prefix adder in an HDL.

Exercise 5.42 An incrementer adds 1 to an N-bit number. Build an 8-bit incrementer using half adders.

Exercise 5.43 Build a 32-bit synchronous *Up/Down counter*. The inputs are *Reset* and *Up*. When *Reset* is 1, the outputs are all 0. Otherwise, when $Up = 1$, the circuit counts up, and when $Up = 0$, the circuit counts down.

Exercise 5.44 Design a 32-bit counter that adds 4 at each clock edge. The counter has reset and clock inputs. Upon reset, the counter output is all 0.

Exercise 5.45 Modify the counter from Exercise 5.44 such that the counter will either increment by 4 or load a new 32-bit value, D, on each clock edge, depending on a control signal *Load*. When $Load = 1$, the counter loads the new value D.

Exercise 5.46 An N-bit *Johnson counter* consists of an N-bit shift register with a reset signal. The output of the shift register (S_{out}) is inverted and fed back to the input (S_{in}). When the counter is reset, all of the bits are cleared to 0.

(a) Show the sequence of outputs, $Q_{3:0}$, produced by a 4-bit Johnson counter starting immediately after the counter is reset.

(b) How many cycles elapse until an N-bit Johnson counter repeats its sequence? Explain.

(c) Design a decimal counter using a 5-bit Johnson counter, ten AND gates, and inverters. The decimal counter has a clock, a reset, and ten one-hot outputs $Y_{9:0}$. When the counter is reset, Y_0 is asserted. On each subsequent cycle, the next output should be asserted. After ten cycles, the counter should repeat. Sketch a schematic of the decimal counter.

(d) What advantages might a Johnson counter have over a conventional counter?

Exercise 5.47 Write the HDL for a 4-bit scannable flip-flop like the one shown in Figure 5.37. Simulate and test your HDL module to prove that it functions correctly.

Exercise 5.48 The English language has a good deal of redundancy that allows us to reconstruct garbled transmissions. Binary data can also be transmitted in redundant form to allow error correction. For example, the number 0 could be coded as 00000 and the number 1 could be coded as 11111. The value could then be sent over a noisy channel that might flip up to two of the bits. The receiver could reconstruct the original data because a 0 will have at least three of the five received bits as 0's; similarly a 1 will have at least three 1's.

(a) Propose an encoding to send 00, 01, 10, or 11 encoded using five bits of information such that all errors that corrupt one bit of the encoded data can be corrected. Hint: the encodings 00000 and 11111 for 00 and 11, respectively, will not work.

(b) Design a circuit that receives your five-bit encoded data and decodes it to 00, 01, 10, or 11, even if one bit of the transmitted data has been changed.

(c) Suppose you wanted to change to an alternative 5-bit encoding. How might you implement your design to make it easy to change the encoding without having to use different hardware?

Exercise 5.49 Flash EEPROM, simply called Flash memory, is a fairly recent invention that has revolutionized consumer electronics. Research and explain how Flash memory works. Use a diagram illustrating the floating gate. Describe how a bit in the memory is programmed. Properly cite your sources.

Exercise 5.50 The extraterrestrial life project team has just discovered aliens living on the bottom of Mono Lake. They need to construct a circuit to classify the aliens by potential planet of origin based on measured features available from the NASA probe: greenness, brownness, sliminess, and ugliness. Careful consultation with xenobiologists leads to the following conclusions:

- If the alien is green and slimy or ugly, brown, and slimy, it might be from Mars.
- If the critter is ugly, brown, and slimy, or green and neither ugly nor slimy, it might be from Venus.
- If the beastie is brown and neither ugly nor slimy or is green and slimy, it might be from Jupiter.

Note that this is an inexact science; for example, a life form which is mottled green and brown and is slimy but not ugly might be from either Mars or Jupiter.

(a) Program a $4 \times 4 \times 3$ PLA to identify the alien. You may use dot notation.

(b) Program a 16×3 ROM to identify the alien. You may use dot notation.

(c) Implement your design in an HDL.

Exercise 5.51 Implement the following functions using a single 16 × 3 ROM. Use dot notation to indicate the ROM contents.

(a) $X = AB + B\overline{C}D + \overline{A}\,\overline{B}$

(b) $Y = AB + BD$

(c) $Z = A + B + C + D$

Exercise 5.52 Implement the functions from Exercise 5.51 using a 4 × 8 × 3 PLA. You may use dot notation.

Exercise 5.53 Specify the size of a ROM that you could use to program each of the following combinational circuits. Is using a ROM to implement these functions a good design choice? Explain why or why not.

(a) a 16-bit adder/subtractor with C_{in} and C_{out}

(b) an 8 × 8 multiplier

(c) a 16-bit priority encoder (see Exercise 2.36)

Exercise 5.54 Consider the ROM circuits in Figure 5.65. For each row, can the circuit in column I be replaced by an equivalent circuit in column II by proper programming of the latter's ROM?

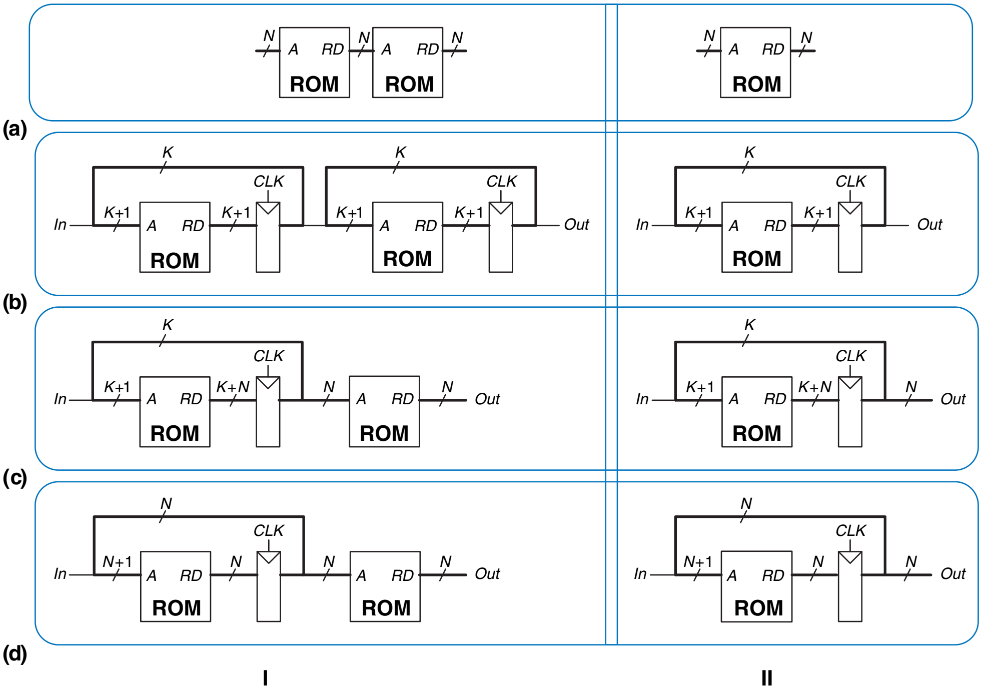


Figure 5.65 ROM circuits

Exercise 5.55 How many Cyclone IV FPGA LEs are required to perform each of the following functions? Show how to configure one or more LEs to perform the function. You should be able to do this by inspection, without performing logic synthesis.

(a) the combinational function from Exercise 2.13(c)

(b) the combinational function from Exercise 2.17(c)

(c) the two-output function from Exercise 2.24

(d) the function from Exercise 2.35

(e) a four-input priority encoder (see Exercise 2.36)

Exercise 5.56 Repeat Exercise 5.55 for the following functions.

(a) an eight-input priority encoder (see Exercise 2.36)

(b) a 3:8 decoder

(c) a 4-bit carry propagate adder (with no carry in or out)

(d) the FSM from Exercise 3.22

(e) the Gray code counter from Exercise 3.27

Exercise 5.57 Consider the Cyclone IV LE shown in Figure 5.58. According to the datasheet, it has the timing specifications given in Table 5.5.

(a) What is the minimum number of Cyclone IV LEs required to implement the FSM of Figure 3.26?

(b) Without clock skew, what is the fastest clock frequency at which this FSM will run reliably?

(c) With 3 ns of clock skew, what is the fastest frequency at which the FSM will run reliably?

Table 5.5 **Cyclone IV timing**

Name	Value (ps)
t_{pcq}, t_{ccq}	199
t_{setup}	76
t_{hold}	0
t_{pd} (per LE)	381
t_{wire} (between LEs)	246
t_{skew}	0

Exercise 5.58 Repeat Exercise 5.57 for the FSM of Figure 3.31(b).

Exercise 5.59 You would like to use an FPGA to implement an M&M sorter with a color sensor and motors to put red candy in one jar and green candy in another. The design is to be implemented as an FSM using a Cyclone IV FPGA. According to the data sheet, the FPGA has timing characteristics shown in Table 5.5. You would like your FSM to run at 100 MHz. What is the maximum number of LEs on the critical path? What is the fastest speed at which the FSM will run?

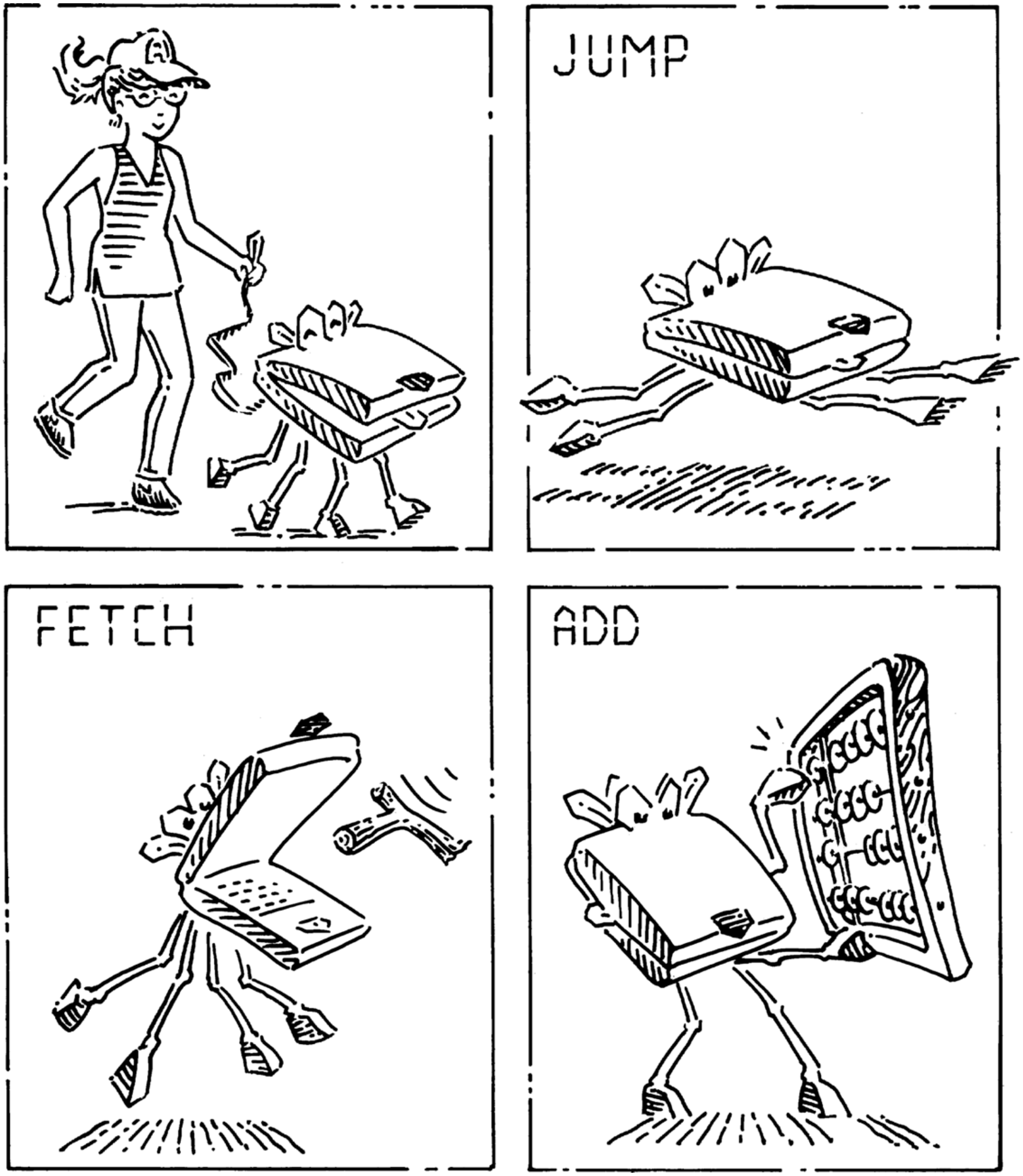
Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 5.1 What is the largest possible result of multiplying two unsigned N-bit numbers?

Question 5.2 *Binary coded decimal* (*BCD*) representation uses four bits to encode each decimal digit. For example, 42_{10} is represented as 01000010_{BCD}. Explain in words why processors might use BCD representation.

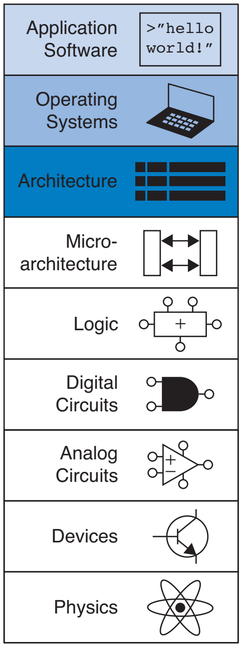
Question 5.3 Design hardware to add two 8-bit unsigned BCD numbers (see Question 5.2). Sketch a schematic for your design, and write an HDL module for the BCD adder. The inputs are A, B, and C_{in}, and the outputs are S and C_{out}. C_{in} and C_{out} are 1-bit carries and A, B, and S are 8-bit BCD numbers.


JUMP
FETCH
ADD

Architecture

6

6.1 Introduction
6.2 Assembly Language
6.3 Machine Language
6.4 Programming
6.5 Addressing Modes
6.6 Lights, Camera, Action: Compiling, Assembling, and Loading
6.7 Odds and Ends*
6.8 Real-World Perspective: x86 Architecture*
6.9 Summary
Exercises
Interview Questions



6.1 INTRODUCTION

The previous chapters introduced digital design principles and building blocks. In this chapter, we jump up a few levels of abstraction to define the *architecture* of a computer. The architecture is the programmer's view of a computer. It is defined by the instruction set (language) and operand locations (registers and memory). Many different architectures exist, such as x86, MIPS, SPARC, and PowerPC.

The first step in understanding any computer architecture is to learn its language. The words in a computer's language are called *instructions*. The computer's vocabulary is called the *instruction set*. All programs running on a computer use the same instruction set. Even complex software applications, such as word processing and spreadsheet applications, are eventually compiled into a series of simple instructions such as add, subtract, and jump. Computer instructions indicate both the operation to perform and the operands to use. The operands may come from memory, from registers, or from the instruction itself.

Computer hardware understands only 1's and 0's, so instructions are encoded as binary numbers in a format called *machine language*. Just as we use letters to encode human language, computers use binary numbers to encode machine language. Microprocessors are digital systems that read and execute machine language instructions. However, humans consider reading machine language to be tedious, so we prefer to represent the instructions in a symbolic format called *assembly language*.

The instruction sets of different architectures are more like different dialects than different languages. Almost all architectures define basic instructions, such as add, subtract, and jump, that operate on memory or registers. Once you have learned one instruction set, understanding others is fairly straightforward.

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00006-9
© 2013 Elsevier, Inc. All rights reserved.

A computer architecture does not define the underlying hardware implementation. Often, many different hardware implementations of a single architecture exist. For example, Intel and Advanced Micro Devices (AMD) both sell various microprocessors belonging to the same x86 architecture. They all can run the same programs, but they use different underlying hardware and therefore offer trade-offs in performance, price, and power. Some microprocessors are optimized for high-performance servers, whereas others are optimized for long battery life in laptop computers. The specific arrangement of registers, memories, ALUs, and other building blocks to form a microprocessor is called the *microarchitecture* and will be the subject of Chapter 7. Often, many different microarchitectures exist for a single architecture.

In this text, we introduce the MIPS architecture that was first developed by John Hennessy and his colleagues at Stanford in the 1980s. MIPS processors are used by, among others, Silicon Graphics, Nintendo, and Cisco. We start by introducing the basic instructions, operand locations, and machine language formats. We then introduce more instructions used in common programming constructs, such as branches, loops, array manipulations, and function calls.

Throughout the chapter, we motivate the design of the MIPS architecture using four principles articulated by Patterson and Hennessy: (1) simplicity favors regularity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises.

What is the best architecture to study when first learning the subject?

Commercially successful architectures such as x86 are satisfying to study because you can use them to write programs on real computers. Unfortunately, many of these architectures are full of warts and idiosyncrasies accumulated over years of haphazard development by different engineering teams, making the architectures difficult to understand and implement.

Many textbooks teach imaginary architectures that are simplified to illustrate the key concepts.

We follow the lead of David Patterson and John Hennessy in their text *Computer Organization and Design* by focusing on the MIPS architecture. Hundreds of millions of MIPS microprocessors have shipped, so the architecture is commercially very important. Yet it is a clean architecture with little odd behavior. At the end of this chapter, we briefly visit the x86 architecture to compare and contrast it with MIPS.

6.2 ASSEMBLY LANGUAGE

Assembly language is the human-readable representation of the computer's native language. Each assembly language instruction specifies both the operation to perform and the operands on which to operate. We introduce simple arithmetic instructions and show how these operations are written in assembly language. We then define the MIPS instruction operands: registers, memory, and constants.

This chapter assumes that you already have some familiarity with a *high-level programming language* such as C, C++, or Java. (These languages are practically identical for most of the examples in this chapter, but where they differ, we will use C.) Appendix C provides an introduction to C for those with little or no prior programming experience.

6.2.1 Instructions

The most common operation computers perform is addition. Code Example 6.1 shows code for adding variables `b` and `c` and writing the result to `a`. The code is shown on the left in a high-level language (using the syntax of C, C++, and Java), and then rewritten on the right in MIPS

Code Example 6.1 ADDITION

High-Level Code

```
a = b + c;
```

MIPS Assembly Code

```
add a, b, c
```

Code Example 6.2 SUBTRACTION

High-Level Code

```
a = b − c;
```

MIPS Assembly Code

```
sub a, b, c
```

assembly language. Note that statements in a C program end with a semicolon.

The first part of the assembly instruction, `add`, is called the *mnemonic* and indicates what operation to perform. The operation is performed on `b` and `c`, the *source operands*, and the result is written to `a`, the *destination operand*.

Code Example 6.2 shows that subtraction is similar to addition. The instruction format is the same as the `add` instruction except for the operation specification, `sub`. This consistent instruction format is an example of the first design principle:

Design Principle 1: Simplicity favors regularity.

Instructions with a consistent number of operands—in this case, two sources and one destination—are easier to encode and handle in hardware. More complex high-level code translates into multiple MIPS instructions, as shown in Code Example 6.3.

In the high-level language examples, single-line comments begin with `//` and continue until the end of the line. Multiline comments begin with `/*` and end with `*/`. In assembly language, only single-line comments are used. They begin with # and continue until the end of the line. The assembly language program in Code Example 6.3 requires a temporary variable `t` to store the intermediate result. Using multiple assembly language instructions

mnemonic (pronounced *ni-mon-ik*) comes from the Greek word μιμνΕσκεστηαι, to remember. The assembly language mnemonic is easier to remember than a machine language pattern of 0's and 1's representing the same operation.

Code Example 6.3 MORE COMPLEX CODE

High-Level Code

```
a = b + c − d;    // single-line comment
                  /* multiple-line
                     comment */
```

MIPS Assembly Code

```
sub t, c, d       # t = c − d
add a, b, t       # a = b + t
```

to perform more complex operations is an example of the second design principle of computer architecture:

Design Principle 2: Make the common case fast.

The MIPS instruction set makes the common case fast by including only simple, commonly used instructions. The number of instructions is kept small so that the hardware required to decode the instruction and its operands can be simple, small, and fast. More elaborate operations that are less common are performed using sequences of multiple simple instructions. Thus, MIPS is a *reduced instruction set computer* (*RISC*) architecture. Architectures with many complex instructions, such as Intel's x86 architecture, are *complex instruction set computers* (*CISC*). For example, x86 defines a "string move" instruction that copies a string (a series of characters) from one part of memory to another. Such an operation requires many, possibly even hundreds, of simple instructions in a RISC machine. However, the cost of implementing complex instructions in a CISC architecture is added hardware and overhead that slows down the simple instructions.

A RISC architecture minimizes the hardware complexity and the necessary instruction encoding by keeping the set of distinct instructions small. For example, an instruction set with 64 simple instructions would need $\log_2 64 = 6$ bits to encode the operation. An instruction set with 256 complex instructions would need $\log_2 256 = 8$ bits of encoding per instruction. In a CISC machine, even though the complex instructions may be used only rarely, they add overhead to all instructions, even the simple ones.

6.2.2 Operands: Registers, Memory, and Constants

An instruction operates on *operands*. In Code Example 6.1 the variables `a`, `b`, and `c` are all operands. But computers operate on 1's and 0's, not variable names. The instructions need a physical location from which to retrieve the binary data. Operands can be stored in registers or memory, or they may be *constants* stored in the instruction itself. Computers use various locations to hold operands in order to optimize for speed and data capacity. Operands stored as constants or in registers are accessed quickly, but they hold only a small amount of data. Additional data must be accessed from memory, which is large but slow. MIPS is called a 32-bit architecture because it operates on 32-bit data. (The MIPS architecture has been extended to 64 bits in commercial products, but we will consider only the 32-bit form in this book.)

Registers

Instructions need to access operands quickly so that they can run fast. But operands stored in memory take a long time to retrieve. Therefore, most architectures specify a small number of registers that hold commonly used operands. The MIPS architecture uses 32 registers, called the *register set* or *register file*. The fewer the registers, the faster they can be accessed. This leads to the third design principle:

Design Principle 3: Smaller is faster.

Looking up information from a small number of relevant books on your desk is a lot faster than searching for the information in the stacks at a library. Likewise, reading data from a small set of registers (for example, 32) is faster than reading it from 1000 registers or a large memory. A small register file is typically built from a small SRAM array (see Section 5.5.3). The SRAM array uses a small decoder and bitlines connected to relatively few memory cells, so it has a shorter critical path than a large memory does.

Code Example 6.4 shows the `add` instruction with register operands. MIPS register names are preceded by the `$` sign. The variables `a`, `b`, and `c` are arbitrarily placed in `$s0`, `$s1`, and `$s2`. The name `$s1` is pronounced "register s1" or "dollar s1". The instruction adds the 32-bit values contained in `$s1` (`b`) and `$s2` (`c`) and writes the 32-bit result to `$s0` (`a`).

MIPS generally stores variables in 18 of the 32 registers: `$s0`–`$s7`, and `$t0`–`$t9`. Register names beginning with `$s` are called *saved* registers. Following MIPS convention, these registers store variables such as `a`, `b`, and `c`. Saved registers have special connotations when they are used with function calls (see Section 6.4.6). Register names beginning with `$t` are called *temporary* registers. They are used for storing temporary variables. Code Example 6.5 shows MIPS assembly code using a temporary register, `$t0`, to store the intermediate calculation of `c – d`.

Code Example 6.4 REGISTER OPERANDS

High-Level Code

```
a = b + c;
```

MIPS Assembly Code

```
# $s0 = a, $s1 = b, $s2 = c
  add $s0, $s1, $s2      # a = b + c
```

Code Example 6.5 TEMPORARY REGISTERS

High-Level Code

```
a = b + c − d;
```

MIPS Assembly Code

```
# $s0 = a, $s1 = b, $s2 = c, $s3 = d

  sub $t0, $s2, $s3      # t = c − d
  add $s0, $s1, $t0      # a = b + t
```

Example 6.1 TRANSLATING HIGH-LEVEL CODE TO ASSEMBLY LANGUAGE

Translate the following high-level code into assembly language. Assume variables `a`–`c` are held in registers `$s0`–`$s2` and `f`–`j` are in `$s3`–`$s7`.

```
a = b - c;
f = (g + h) − (i + j);
```

Solution: The program uses four assembly language instructions.

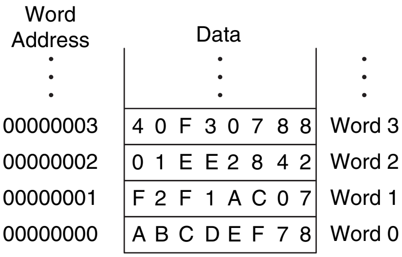
```
# MIPS assembly code
# $s0 = a, $s1 = b, $s2 = c, $s3 = f, $s4 = g, $s5 = h
# $s6 = i, $s7 = j
  sub $s0, $s1, $s2   # a = b - c
  add $t0, $s4, $s5   # $t0 = g + h
  add $t1, $s6, $s7   # $t1 = i + j
  sub $s3, $t0, $t1   # f = (g + h) - (i + j)
```

The Register Set

The **MIPS** architecture defines 32 registers. Each register has a name and a number ranging from 0 to 31. Table 6.1 lists the name, number, and use for each register. `$0` always contains the value 0 because this constant is so frequently used in computer programs. We have also discussed the `$s` and `$t` registers. The remaining registers will be described throughout this chapter.

Table 6.1 MIPS register set

Name	Number	Use
`$0`	0	the constant value 0
`$at`	1	assembler temporary
`$v0`–`$v1`	2–3	function return value
`$a0`–`$a3`	4–7	function arguments
`$t0`–`$t7`	8–15	temporary variables
`$s0`–`$s7`	16–23	saved variables
`$t8`–`$t9`	24–25	temporary variables
`$k0`–`$k1`	26–27	operating system (OS) temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	function return address



Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Figure 6.1 **Word-addressable memory**

Memory

If registers were the only storage space for operands, we would be confined to simple programs with no more than 32 variables. However, data can also be stored in memory. When compared to the register file, memory has many data locations, but accessing it takes a longer amount of time. Whereas the register file is small and fast, memory is large and slow. For this reason, commonly used variables are kept in registers. By using a combination of memory and registers, a program can access a large amount of data fairly quickly. As described in Section 5.5, memories are organized as an array of data words. The MIPS architecture uses 32-bit memory addresses and 32-bit data words.

MIPS uses a byte-addressable memory. That is, each byte in memory has a unique address. However, for explanation purposes only, we first introduce a word-addressable memory, and afterward describe the MIPS byte-addressable memory.

Figure 6.1 shows a memory array that is *word-addressable*. That is, each 32-bit data word has a unique 32-bit address. Both the 32-bit word address and the 32-bit data value are written in hexadecimal in Figure 6.1. For example, data 0xF2F1AC07 is stored at memory address 1. Hexadecimal constants are written with the prefix 0x. By convention, memory is drawn with low memory addresses toward the bottom and high memory addresses toward the top.

MIPS uses the *load word* instruction, `lw`, to read a data word from memory into a register. Code Example 6.6 loads memory word 1 into `$s3`.

The `lw` instruction specifies the *effective address* in memory as the sum of a *base address* and an *offset*. The base address (written in parentheses in the instruction) is a register. The offset is a constant (written before the parentheses). In Code Example 6.6, the base address is `$0`, which holds the value 0, and the offset is 1, so the `lw` instruction reads from memory

---

Code Example 6.6 READING WORD-ADDRESSABLE MEMORY

Assembly Code

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  lw $s3, 1($0)     # read memory word 1 into $s3
```

Code Example 6.7 WRITING WORD-ADDRESSABLE MEMORY

Assembly Code

```
# This assembly code (unlike MIPS) assumes word-addressable memory
  sw   $s7, 5($0)      # write $s7 to memory word 5
```

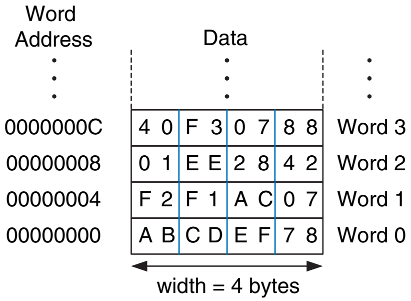

Word Address
Data
0000000C
00000008
00000004
00000000
4 0 F 3 0 7 8 8
0 1 E E 2 8 4 2
F 2 F 1 A C 0 7
A B C D E F 7 8
Word 3
Word 2
Word 1
Word 0
width = 4 bytes

Figure 6.2 **Byte-addressable memory**

address (`$0` + `1`) = `1`. After the load word instruction (`lw`) is executed, `$s3` holds the value 0xF2F1AC07, which is the data value stored at memory address 1 in Figure 6.1.

Similarly, MIPS uses the *store word* instruction, `sw`, to write a data word from a register into memory. Code Example 6.7 writes the contents of register `$s7` into memory word 5. These examples have used `$0` as the base address for simplicity, but remember that any register can be used to supply the base address.

The previous two code examples have shown a computer architecture with a word-addressable memory. The MIPS memory model, however, is byte-addressable, *not* word-addressable. Each data byte has a unique address. A 32-bit word consists of four 8-bit bytes. So each word address is a multiple of 4, as shown in Figure 6.2. Again, both the 32-bit word address and the data value are given in hexadecimal.

Code Example 6.8 shows how to read and write words in the MIPS byte-addressable memory. The word address is four times the word number. The MIPS assembly code reads words 0, 2, and 3 and writes words 1, 8, and 100. The offset can be written in decimal or hexadecimal.

The MIPS architecture also provides the `lb` and `sb` instructions that load and store single bytes in memory rather than words. They are similar to `lw` and `sw` and will be discussed further in Section 6.4.5.

Byte-addressable memories are organized in a *big-endian* or *little-endian* fashion, as shown in Figure 6.3. In both formats, the most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. In big-endian machines, bytes are numbered starting with 0 at the big (most

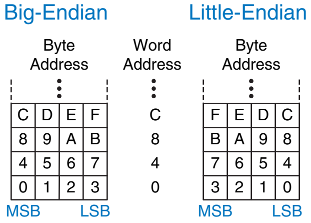

Big-Endian
Little-Endian
Byte Address
Word Address
Byte Address
C D E F
8 9 A B
4 5 6 7
0 1 2 3
C
8
4
0
F E D C
B A 9 8
7 6 5 4
3 2 1 0
MSB
LSB
MSB
LSB

Figure 6.3 **Big- and little-endian memory addressing**

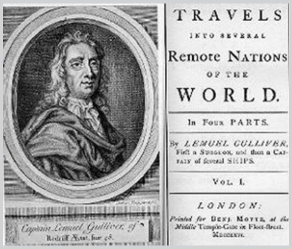
Code Example 6.8 ACCESSING BYTE-ADDRESSABLE MEMORY

MIPS Assembly Code

```
lw $s0, 0($0)       # read data word 0 (0xABCDEF78) into $s0
lw $s1, 8($0)       # read data word 2 (0x01EE2842) into $s1
lw $s2, 0xC($0)     # read data word 3 (0x40F30788) into $s2
sw $s3, 4($0)       # write $s3 to data word 1
sw $s4, 0x20($0)    # write $s4 to data word 8
sw $s5, 400($0)     # write $s5 to data word 100
```

---

significant) end. In little-endian machines, bytes are numbered starting with 0 at the little (least significant) end. Word addresses are the same in both formats and refer to the same four bytes. Only the addresses of bytes within a word differ.



The terms big-endian and little-endian come from Jonathan Swift's *Gulliver's Travels*, first published in 1726 under the pseudonym of Isaac Bickerstaff. In his stories the Lilliputian king required his citizens (the Little-Endians) to break their eggs on the little end. The Big-Endians were rebels who broke their eggs on the big end.

The terms were first applied to computer architectures by Danny Cohen in his paper "On Holy Wars and a Plea for Peace" published on April Fools Day, 1980 (*USC/ISI IEN 137*). (Photo courtesy of The Brotherton Collection, IEEDS University Library.)

Example 6.2 BIG- AND LITTLE-ENDIAN MEMORY

Suppose that `$s0` initially contains 0x23456789. After the following program is run on a big-endian system, what value does `$s0` contain? In a little-endian system? `lb $s0, 1($0)` loads the data at byte address (`1` + `$0`) = `1` into the least significant byte of `$s0`. `lb` is discussed in detail in Section 6.4.5.

```
sw $s0, 0($0)

lb $s0, 1($0)
```

Solution: Figure 6.4 shows how big- and little-endian machines store the value 0x23456789 in memory word 0. After the load byte instruction, `lb $s0, 1($0)`, `$s0` would contain 0x00000045 on a big-endian system and 0x00000067 on a little-endian system.

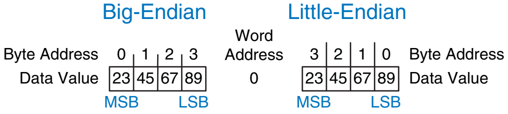


Figure 6.4 **Big-endian and little-endian data storage**

IBM's PowerPC (formerly found in Macintosh computers) uses big-endian addressing. Intel's x86 architecture (found in PCs) uses little-endian addressing. Some MIPS processors are little-endian, and some are big-endian.[1] The choice of endianness is completely arbitrary but leads to hassles when sharing data between big-endian and little-endian computers. In examples in this text, we will use little-endian format whenever byte ordering matters.

---

[1] SPIM, the MIPS simulator that comes with this text, uses the endianness of the machine it is run on. For example, when using SPIM on an Intel x86 machine, the memory is little-endian. With an older Macintosh or Sun SPARC machine, memory is big-endian.

In the MIPS architecture, word addresses for `lw` and `sw` must be *word aligned*. That is, the address must be divisible by 4. Thus, the instruction `lw $s0, 7($0)` is an illegal instruction. Some architectures, such as x86, allow non-word-aligned data reads and writes, but MIPS requires strict alignment for simplicity. Of course, byte addresses for load byte and store byte, `lb` and `sb`, need not be word aligned.

Constants/Immediates

Load word and store word, `lw` and `sw`, also illustrate the use of *constants* in MIPS instructions. These constants are called *immediates*, because their values are immediately available from the instruction and do not require a register or memory access. Add immediate, `addi`, is another common MIPS instruction that uses an immediate operand. `addi` adds the immediate specified in the instruction to a value in a register, as shown in Code Example 6.9.

Code Example 6.9 IMMEDIATE OPERANDS

High-Level Code

```
a = a + 4;
b = a − 12;
```

MIPS Assembly Code

```
# $s0 = a, $s1 = b
  addi $s0, $s0, 4          # a = a + 4
  addi $s1, $s0, −12        # b = a − 12
```

The immediate specified in an instruction is a 16-bit two's complement number in the range [–32,768, 32,767]. Subtraction is equivalent to adding a negative number, so, in the interest of simplicity, there is no `subi` instruction in the MIPS architecture.

Recall that the `add` and `sub` instructions use three register operands. But the `lw`, `sw`, and `addi` instructions use two register operands and a constant. Because the instruction formats differ, `lw` and `sw` instructions violate design principle 1: simplicity favors regularity. However, this issue allows us to introduce the last design principle:

Design Principle 4: Good design demands good compromises.

A single instruction format would be simple but not flexible. The MIPS instruction set makes the compromise of supporting three instruction formats. One format, used for instructions such as `add` and `sub`, has three register operands. Another, used for instructions such as `lw` and `addi`, has two register operands and a 16-bit immediate. A third, to be discussed later, has a 26-bit immediate and no registers. The next section discusses the three MIPS instruction formats and shows how they are encoded into binary.

6.3 MACHINE LANGUAGE

Assembly language is convenient for humans to read. However, digital circuits understand only 1's and 0's. Therefore, a program written in assembly language is translated from mnemonics to a representation using only 1's and 0's called *machine language*.

MIPS uses 32-bit instructions. Again, simplicity favors regularity, and the most regular choice is to encode all instructions as words that can be stored in memory. Even though some instructions may not require all 32 bits of encoding, variable-length instructions would add too much complexity. Simplicity would also encourage a single instruction format, but, as already mentioned, that is too restrictive. MIPS makes the compromise of defining three instruction formats: R-type, I-type, and J-type. This small number of formats allows for some regularity among all the types, and thus simpler hardware, while also accommodating different instruction needs, such as the need to encode large constants in the instruction. *R-type* instructions operate on three registers. *I-type* instructions operate on two registers and a 16-bit immediate. *J-type* (jump) instructions operate on one 26-bit immediate. We introduce all three formats in this section but leave the discussion of J-type instructions for Section 6.4.2.

6.3.1 R-Type Instructions

The name R-type is short for *register-type*. R-type instructions use three registers as operands: two as sources, and one as a destination. Figure 6.5 shows the R-type machine instruction format. The 32-bit instruction has six fields: `op`, `rs`, `rt`, `rd`, `shamt`, and `funct`. Each field is five or six bits, as indicated.

The operation the instruction performs is encoded in the two fields highlighted in blue: `op` (also called `opcode` or operation code) and `funct` (also called the function). All R-type instructions have an opcode of 0. The specific R-type operation is determined by the `funct` field. For example, the `opcode` and `funct` fields for the `add` instruction are 0 (000000_2) and 32 (100000_2), respectively. Similarly, the `sub` instruction has an `opcode` and `funct` field of 0 and 34.

The operands are encoded in the three fields: `rs`, `rt`, and `rd`. The first two registers, `rs` and `rt`, are the source registers; `rd` is the destination

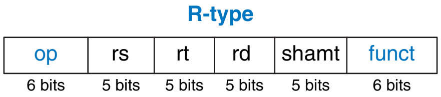


Figure 6.5 **R-type machine instruction format**

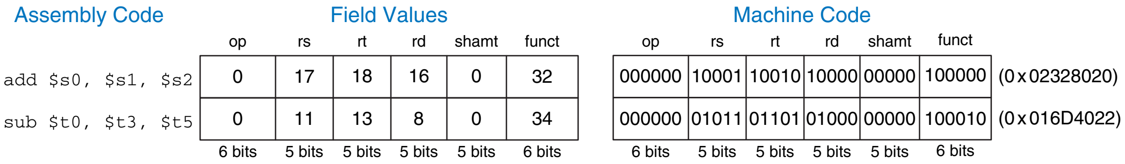
`rs` is short for "register source." `rt` comes after `rs` alphabetically and usually indicates the second register source.

register. The fields contain the register numbers that were given in Table 6.1. For example, `$s0` is register 16.

The fifth field, `shamt`, is used only in shift operations. In those instructions, the binary value stored in the 5-bit `shamt` field indicates the amount to shift. For all other R-type instructions, `shamt` is 0.

Figure 6.6 shows the machine code for the R-type instructions `add` and `sub`. Notice that the destination is the first register in an assembly language instruction, but it is the third register field (`rd`) in the machine language instruction. For example, the assembly instruction `add $s0, $s1, $s2` has `rs` = `$s1` (17), `rt` = `$s2` (18), and `rd` = `$s0` (16).

For MIPS instructions used in this book, Tables B.1 and B.2 in Appendix B define the opcode values for all instructions and the `funct` field values for R-type instructions.



Assembly Code	op	rs	rt	rd	shamt	funct
`add $s0, $s1, $s2`	0	17	18	16	0	32
`sub $t0, $t3, $t5`	0	11	13	8	0	34
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

op	rs	rt	rd	shamt	funct	Machine Code
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

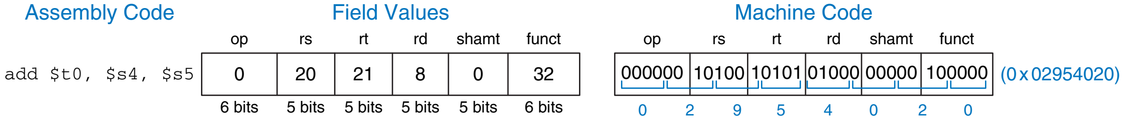
Figure 6.6 Machine code for R-type instructions

Example 6.3 TRANSLATING ASSEMBLY LANGUAGE TO MACHINE LANGUAGE

Translate the following assembly language statement into machine language.

```
add $t0, $s4, $s5
```

Solution: According to Table 6.1, `$t0`, `$s4`, and `$s5` are registers 8, 20, and 21. According to Tables B.1 and B.2, `add` has an `opcode` of 0 and a `funct` code of 32. Thus, the fields and machine code are given in Figure 6.7. The easiest way to write the machine language in hexadecimal is to first write it in binary, then look at consecutive groups of four bits, which correspond to hexadecimal digits (indicated in blue). Hence, the machine language instruction is 0x02954020.



Assembly Code	op	rs	rt	rd	shamt	funct
`add $t0, $s4, $s5`	0	20	21	8	0	32
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

op	rs	rt	rd	shamt	funct	Machine Code
000000	10100	10101	01000	00000	100000	(0x02954020)

Figure 6.7 Machine code for the R-type instruction of Example 6.3

6.3.2 I-Type Instructions

The name I-type is short for *immediate-type*. I-type instructions use two register operands and one immediate operand. Figure 6.8 shows the I-type machine instruction format. The 32-bit instruction has four fields: `op`, `rs`, `rt`, and `imm`. The first three fields, `op`, `rs`, and `rt`, are like those of R-type instructions. The `imm` field holds the 16-bit immediate.

The operation is determined solely by the `opcode`, highlighted in blue. The operands are specified in the three fields `rs`, `rt`, and `imm`. `rs` and `imm` are always used as source operands. `rt` is used as a destination for some instructions (such as `addi` and `lw`) but as another source for others (such as `sw`).

Figure 6.9 shows several examples of encoding I-type instructions. Recall that negative immediate values are represented using 16-bit two's complement notation. `rt` is listed first in the assembly language instruction when it is used as a destination, but it is the second register field in the machine language instruction.

I-type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Figure 6.8 **I-type instruction format**

Assembly Code	Field Values				Machine Code				
	op	rs	rt	imm	op	rs	rt	imm	
`addi $s0, $s1, 5`	8	17	16	5	001000	10001	10000	0000 0000 0000 0101	(0x22300005)
`addi $t0, $s3, -12`	8	19	8	–12	001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
`lw   $t2, 32($0)`	35	0	10	32	100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
`sw   $s1,  4($t1)`	43	9	17	4	101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
	6 bits	5 bits	5 bits	16 bits	6 bits	5 bits	5 bits	16 bits	

Figure 6.9 **Machine code for I-type instructions**

Example 6.4 TRANSLATING I-TYPE ASSEMBLY INSTRUCTIONS INTO MACHINE CODE

Translate the following I-type instruction into machine code.

```
lw $s3, −24($s4)
```

Solution: According to Table 6.1, `$s3` and `$s4` are registers 19 and 20, respectively. Table B.1 indicates that `lw` has an `opcode` of 35. `rs` specifies the base address, `$s4`, and `rt` specifies the destination register, `$s3`. The immediate, `imm`, encodes the 16-bit offset, −24. The fields and machine code are given in Figure 6.10.

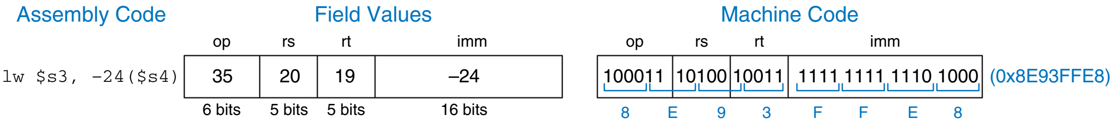


Figure 6.10 Machine code for an I-type instruction

I-type instructions have a 16-bit immediate field, but the immediates are used in 32-bit operations. For example, `lw` adds a 16-bit offset to a 32-bit base register. What should go in the upper half of the 32 bits? For positive immediates, the upper half should be all 0's, but for negative immediates, the upper half should be all 1's. Recall from Section 1.4.6 that this is called *sign extension*. An N-bit two's complement number is sign-extended to an M-bit number ($M > N$) by copying the sign bit (most significant bit) of the N-bit number into all of the upper bits of the M-bit number. Sign-extending a two's complement number does not change its value.

Most MIPS instructions sign-extend the immediate. For example, `addi`, `lw`, and `sw` do sign extension to support both positive and negative immediates. An exception to this rule is that logical operations (`andi`, `ori`, `xori`) place 0's in the upper half; this is called *zero extension* rather than sign extension. Logical operations are discussed further in Section 6.4.1.

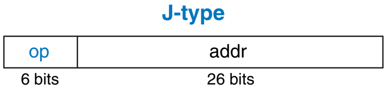
6.3.3 J-Type Instructions

The name J-type is short for *jump-type*. This format is used only with jump instructions (see Section 6.4.2). This instruction format uses a single 26-bit address operand, `addr`, as shown in Figure 6.11. Like other formats, J-type instructions begin with a 6-bit `opcode`. The remaining bits are used to specify an address, `addr`. Further discussion and machine code examples of J-type instructions are given in Sections 6.4.2 and 6.5.

6.3.4 Interpreting Machine Language Code

To interpret machine language, one must decipher the fields of each 32-bit instruction word. Different instructions use different formats, but all formats start with a 6-bit `opcode` field. Thus, the best place to begin is to look at the `opcode`. If it is 0, the instruction is R-type; otherwise it is I-type or J-type.

Figure 6.11 J-type instruction format



Example 6.5 TRANSLATING MACHINE LANGUAGE TO ASSEMBLY LANGUAGE

Translate the following machine language code into assembly language.

```
0x2237FFF1
0x02F34022
```

Solution: First, we represent each instruction in binary and look at the six most significant bits to find the `opcode` for each instruction, as shown in Figure 6.12. The `opcode` determines how to interpret the rest of the bits. The `opcodes` are 001000_2 (8_{10}) and 000000_2 (0_{10}), indicating an `addi` and R-type instruction, respectively. The `funct` field of the R-type instruction is 100010_2 (34_{10}), indicating that it is a `sub` instruction. Figure 6.12 shows the assembly code equivalent of the two machine instructions.

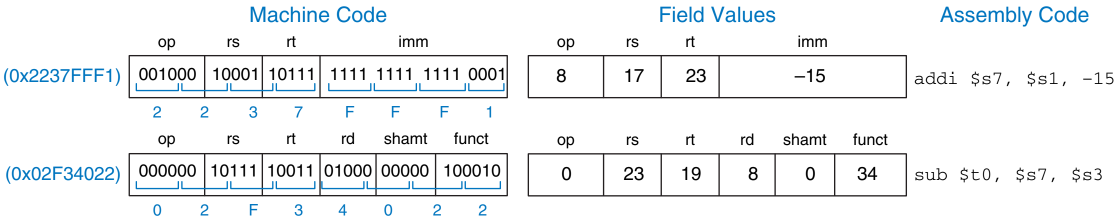

Machine Code
Field Values
Assembly Code
op
rs
rt
imm
(0x2237FFF1)
001000
10001
10111
1111 1111 1111 0001
2 2 3 7 F F F 1
8
17
23
–15
addi $s7, $s1, -15
op
rs
rt
rd
shamt
funct
(0x02F34022)
000000
10111
10011
01000
00000
100010
0 2 F 3 4 0 2 2
0
23
19
8
0
34
sub $t0, $s7, $s3

Figure 6.12 Machine code to assembly code translation

6.3.5 The Power of the Stored Program

A program written in machine language is a series of 32-bit numbers representing the instructions. Like other binary numbers, these instructions can be stored in memory. This is called the *stored program* concept, and it is a key reason why computers are so powerful. Running a different program does not require large amounts of time and effort to reconfigure or rewire hardware; it only requires writing the new program to memory. Instead of dedicated hardware, the stored program offers *general purpose* computing. In this way, a computer can execute applications ranging from a calculator to a word processor to a video player simply by changing the stored program.

Instructions in a stored program are retrieved, or *fetched*, from memory and executed by the processor. Even large, complex programs are simplified to a series of memory reads and instruction executions.

Figure 6.13 shows how machine instructions are stored in memory. In MIPS programs, the instructions are normally stored starting at address 0x00400000. Remember that MIPS memory is byte-addressable, so 32-bit (4-byte) instruction addresses advance by 4 bytes, not 1.

Assembly Code	Machine Code
`lw   $t2, 32($0)`	`0x8C0A0020`
`add  $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub  $t0, $t3, $t5`	`0x016D4022`

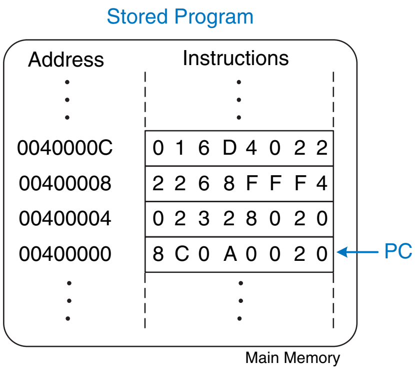


Figure 6.13 **Stored program**

To run or *execute* the stored program, the processor fetches the instructions from memory sequentially. The fetched instructions are then decoded and executed by the digital hardware. The address of the current instruction is kept in a 32-bit register called the *program counter* (`PC`). The `PC` is separate from the 32 registers shown previously in Table 6.1.

To execute the code in Figure 6.13, the operating system sets the `PC` to address 0x00400000. The processor reads the instruction at that memory address and executes the instruction, 0x8C0A0020. The processor then increments the `PC` by 4 to 0x00400004, fetches and executes that instruction, and repeats.

The *architectural state* of a microprocessor holds the state of a program. For MIPS, the architectural state consists of the register file and `PC`. If the operating system saves the architectural state at some point in the program, it can interrupt the program, do something else, then restore the state such that the program continues properly, unaware that it was ever interrupted. The architectural state is also of great importance when we build a microprocessor in Chapter 7.

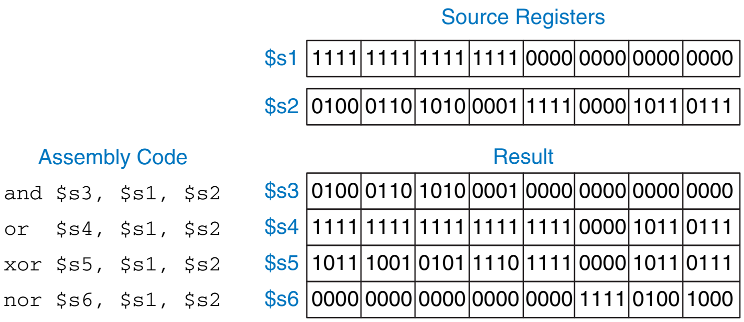
Ada Lovelace, 1815–1852. Wrote the first computer program. It calculated the Bernoulli numbers using Charles Babbage's Analytical Engine. She was the only legitimate child of the poet Lord Byron.

6.4 PROGRAMMING

Software languages such as C or Java are called high-level programming languages because they are written at a more abstract level than assembly language. Many high-level languages use common software constructs such as arithmetic and logical operations, `if/else` statements, `for` and `while` loops, array indexing, and function calls. See Appendix C for more examples of these constructs in C. In this section, we explore how to translate these high-level constructs into MIPS assembly code.

6.4.1 Arithmetic/Logical Instructions

The MIPS architecture defines a variety of arithmetic and logical instructions. We introduce these instructions briefly here because they are necessary to implement higher-level constructs.



Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code	Result								
`and $s3, $s1, $s2`	$s3	0100	0110	1010	0001	0000	0000	0000	0000
`or  $s4, $s1, $s2`	$s4	1111	1111	1111	1111	1111	0000	1011	0111
`xor $s5, $s1, $s2`	$s5	1011	1001	0101	1110	1111	0000	1011	0111
`nor $s6, $s1, $s2`	$s6	0000	0000	0000	0000	0000	1111	0100	1000

Figure 6.14 **Logical operations**

Logical Instructions

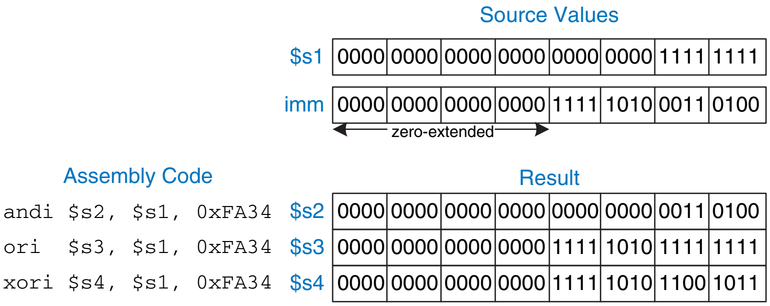
MIPS logical operations include `and`, `or`, `xor`, and `nor`. These R-type instructions operate bit-by-bit on two source registers and write the result to the destination register. Figure 6.14 shows examples of these operations on the two source values 0xFFFF0000 and 0x46A1F0B7. The figure shows the values stored in the destination register, `rd`, after the instruction executes.

The `and` instruction is useful for *masking* bits (i.e., forcing unwanted bits to 0). For example, in Figure 6.14, 0xFFFF0000 AND 0x46A1F0B7 = 0x46A10000. The `and` instruction masks off the bottom two bytes and places the unmasked top two bytes of `$s2`, 0x46A1, in `$s3`. Any subset of register bits can be masked.

The `or` instruction is useful for combining bits from two registers. For example, 0x347A0000 OR 0x000072FC = 0x347A72FC, a combination of the two values.

MIPS does not provide a NOT instruction, but A NOR $0 = NOT A, so the NOR instruction can substitute.

Logical operations can also operate on immediates. These I-type instructions are `andi`, `ori`, and `xori`. `nori` is not provided, because the same functionality can be easily implemented using the other instructions, as will be explored in Exercise 6.16. Figure 6.15 shows examples of the `andi`, `ori`, and `xori` instructions. The figure gives the values of the source



Source Values

$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

Assembly Code	Result								
`andi $s2, $s1, 0xFA34`	$s2	0000	0000	0000	0000	0000	0000	0011	0100
`ori  $s3, $s1, 0xFA34`	$s3	0000	0000	0000	0000	1111	1010	1111	1111
`xori $s4, $s1, 0xFA34`	$s4	0000	0000	0000	0000	1111	1010	1100	1011

Figure 6.15 **Logical operations with immediates**

register and immediate and the value of the destination register `rt` after the instruction executes. Because these instructions operate on a 32-bit value from a register and a 16-bit immediate, they first zero-extend the immediate to 32 bits.

Shift Instructions

Shift instructions shift the value in a register left or right by up to 31 bits. Shift operations multiply or divide by powers of two. MIPS shift operations are `sll` (shift left logical), `srl` (shift right logical), and `sra` (shift right arithmetic).

As discussed in Section 5.2.5, left shifts always fill the least significant bits with 0's. However, right shifts can be either logical (0's shift into the most significant bits) or arithmetic (the sign bit shifts into the most significant bits). Figure 6.16 shows the machine code for the R-type instructions `sll`, `srl`, and `sra`. `rt` (i.e., `$s1`) holds the 32-bit value to be shifted, and `shamt` gives the amount by which to shift (4). The shifted result is placed in `rd`.

Figure 6.17 shows the register values for the shift instructions `sll`, `srl`, and `sra`. Shifting a value left by N is equivalent to multiplying it by 2^N. Likewise, arithmetically shifting a value right by N is equivalent to dividing it by 2^N, as discussed in Section 5.2.5.

MIPS also has variable-shift instructions: `sllv` (shift left logical variable), `srlv` (shift right logical variable), and `srav` (shift right arithmetic variable). Figure 6.18 shows the machine code for these instructions. Variable-shift assembly instructions are of the form `sllv rd, rt, rs`. The order of `rt` and `rs` is reversed from most R-type instructions. `rt` (`$s1`) holds the value to be shifted, and the five least significant bits of `rs` (`$s2`) give the amount to shift. The shifted result is placed in `rd`, as before. The `shamt`

Assembly Code	Field Values						Machine Code						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	
`sll $t0, $s1, 4`	0	0	17	8	4	0	000000	00000	10001	01000	00100	000000	(0x00114100)
`srl $s2, $s1, 4`	0	0	17	18	4	2	000000	00000	10001	10010	00100	000010	(0x00119102)
`sra $s3, $s1, 4`	0	0	17	19	4	3	000000	00000	10001	10011	00100	000011	(0x00119903)
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Figure 6.16 Shift instruction machine code

Source Values

$s1	1111	0011	0000	0000	0000	0010	1010	1000
shamt								00100

Assembly Code	Result								
`sll $t0, $s1, 4`	$t0	0011	0000	0000	0000	0010	1010	1000	0000
`srl $s2, $s1, 4`	$s2	0000	1111	0011	0000	0000	0000	0010	1010
`sra $s3, $s1, 4`	$s3	1111	1111	0011	0000	0000	0000	0010	1010

Figure 6.17 Shift operations

Assembly Code	Field Values						Machine Code						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	
sllv $s3, $s1, $s2	0	18	17	19	0	4	000000	10010	10001	10011	00000	000100	(0x02519804)
srlv $s4, $s1, $s2	0	18	17	20	0	6	000000	10010	10001	10100	00000	000110	(0x0251A006)
srav $s5, $s1, $s2	0	18	17	21	0	7	000000	10010	10001	10101	00000	000111	(0x0251A807)
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Figure 6.18 **Variable-shift instruction machine code**

Source Values

$s1	1111	0011	0000	0100	0000	0010	1010	1000
$s2	0000	0000	0000	0000	0000	0000	0000	**1000**

Assembly Code	Result								
sllv $s3, $s1, $s2	$s3	0000	0100	0000	0010	1010	1000	0000	0000
srlv $s4, $s1, $s2	$s4	0000	0000	1111	0011	0000	0100	0000	0010
srav $s5, $s1, $s2	$s5	1111	1111	1111	0011	0000	0100	0000	0010

Figure 6.19 **Variable-shift operations**

field is ignored and should be all 0's. Figure 6.19 shows register values for each type of variable-shift instruction.

Generating Constants

The `addi` instruction is helpful for assigning 16-bit constants, as shown in Code Example 6.10.

Code Example 6.10 16-BIT CONSTANT

High-Level Code

```
int a = 0x4f3c;
```

MIPS Assembly Code

```
# $s0 = a
  addi $s0, $0, 0x4f3c      # a = 0x4f3c
```

To assign 32-bit constants, use a load upper immediate instruction (`lui`) followed by an or immediate (`ori`) instruction as shown in Code Example 6.11. `lui` loads a 16-bit immediate into the upper half of a register and sets the lower half to 0. As mentioned earlier, `ori` merges a 16-bit immediate into the lower half.

The `int` data type in C refers to a word of data representing a two's complement integer. MIPS uses 32-bit words, so an `int` represents a number in the range $[-2^{31}, 2^{31}-1]$.

Code Example 6.11 32-BIT CONSTANT

High-Level Code

```
int a = 0x6d5e4f3c;
```

MIPS Assembly Code

```
# $s0 = a
  lui $s0, 0x6d5e           # a = 0x6d5e0000
  ori $s0, $s0, 0x4f3c      # a = 0x6d5e4f3c
```

Multiplication and Division Instructions*

Multiplication and division are somewhat different from other arithmetic operations. Multiplying two 32-bit numbers produces a 64-bit product. Dividing two 32-bit numbers produces a 32-bit quotient and a 32-bit remainder.

`hi` and `lo` are not among the usual 32 MIPS registers, so special instructions are needed to access them. `mfhi $s2` (move from `hi`) copies the value in `hi` to `$s2`. `mflo $s3` (move from lo) copies the value in `lo` to `$s3`. `hi` and `lo` are technically part of the architectural state; however, we generally ignore these registers in this book.

The MIPS architecture has two special-purpose registers, `hi` and `lo`, which are used to hold the results of multiplication and division. `mult $s0, $s1` multiplies the values in `$s0` and `$s1`. The 32 most significant bits of the product are placed in `hi` and the 32 least significant bits are placed in `lo`. Similarly, `div $s0, $s1` computes `$s0/$s1`. The quotient is placed in `lo` and the remainder is placed in `hi`.

MIPS provides another multiply instruction that produces a 32-bit result in a general purpose register. `mul $s1, $s2, $s3` multiplies the values in `$s2` and `$s3` and places the 32-bit result in `$s1`.

6.4.2 Branching

An advantage of a computer over a calculator is its ability to make decisions. A computer performs different tasks depending on the input. For example, `if/else` statements, `switch/case` statements, `while` loops, and `for` loops all conditionally execute code depending on some test.

To sequentially execute instructions, the program counter increments by 4 after each instruction. *Branch* instructions modify the program counter to skip over sections of code or to repeat previous code. *Conditional branch* instructions perform a test and branch only if the test is TRUE. *Unconditional branch* instructions, called *jumps*, always branch.

Conditional Branches

The MIPS instruction set has two conditional branch instructions: branch if equal (`beq`) and branch if not equal (`bne`). `beq` branches when the values in two registers are equal, and `bne` branches when they are not equal. Code Example 6.12 illustrates the use of `beq`. Note that branches are written as `beq rs, rt, imm`, where `rs` is the first source register. This order is reversed from most I-type instructions.

When the program in Code Example 6.12 reaches the branch if equal instruction (`beq`), the value in `$s0` is equal to the value in `$s1`, so the branch is *taken*. That is, the next instruction executed is the `add` instruction just after the *label* called `target`. The two instructions directly after the branch and before the label are not executed.[2]

Assembly code uses labels to indicate instruction locations in the program. When the assembly code is translated into machine code, these

---

[2] In practice, because of pipelining (discussed in Chapter 7), MIPS processors have a *branch delay slot*. This means that the instruction immediately after a branch or jump is always executed. This idiosyncracy is ignored in MIPS assembly code in this chapter.

Code Example 6.12 CONDITIONAL BRANCHING USING `beq`

MIPS Assembly Code

```
  addi  $s0, $0, 4        # $s0 = 0 + 4 = 4
  addi  $s1, $0, 1        # $s1 = 0 + 1 = 1
  sll   $s1, $s1, 2       # $s1 = 1 << 2 = 4
  beq   $s0, $s1, target  # $s0 == $s1, so branch is taken
  addi  $s1, $s1, 1       # not executed
  sub   $s1, $s1, $s0     # not executed

target:
  add   $s1, $s1, $s0     # $s1 = 4 + 4 = 8
```

labels are translated into instruction addresses (see Section 6.5). MIPS assembly labels are followed by a colon (`:`) and cannot use reserved words, such as instruction mnemonics. Most programmers indent their instructions but not the labels, to help make labels stand out.

Code Example 6.13 shows an example using the branch if not equal instruction (`bne`). In this case, the branch is *not taken* because `$s0` is equal to `$s1`, and the code continues to execute directly after the `bne` instruction. All instructions in this code snippet are executed.

Code Example 6.13 CONDITIONAL BRANCHING USING `bne`

MIPS Assembly Code

```
  addi  $s0, $0, 4        # $s0 = 0 + 4 = 4
  addi  $s1, $0, 1        # $s1 = 0 + 1 = 1
  sll   $s1, $s1, 2       # $s1 = 1 << 2 = 4
  bne   $s0, $s1, target  # $s0 == $s1, so branch is not taken
  addi  $s1, $s1, 1       # $s1 = 4 + 1 = 5
  sub   $s1, $s1, $s0     # $s1 = 5 – 4 = 1

target:
  add   $s1, $s1, $s0     # $s1 = 1 + 4 = 5
```

Jump

A program can unconditionally branch, or *jump*, using the three types of jump instructions: jump (`j`), jump and link (`jal`), and jump register (`jr`). Jump (`j`) jumps directly to the instruction at the specified label. Jump and link (`jal`) is similar to `j` but is used by functions to save a return address, as will be discussed in Section 6.4.6. Jump register (`jr`) jumps to the address held in a register. Code Example 6.14 shows the use of the jump instruction (`j`).

`j` and `jal` are J-type instructions. `jr` is an R-type instruction that uses only the `rs` operand.

After the `j target` instruction, the program in Code Example 6.14 unconditionally continues executing the `add` instruction at the label `target`. All of the instructions between the jump and the label are skipped.

Code Example 6.14 UNCONDITIONAL BRANCHING USING `j`

MIPS Assembly Code

```
  addi  $s0, $0, 4      # $s0 = 4
  addi  $s1, $0, 1      # $s1 = 1
  j     target          # jump to target
  addi  $s1, $s1, 1     # not executed
  sub   $s1, $s1, $s0   # not executed
target:
  add   $s1, $s1, $s0   # $s1 = 1 + 4 = 5
```

Code Example 6.15 UNCONDITIONAL BRANCHING USING `jr`

MIPS Assembly Code

```
0x00002000  addi  $s0, $0, 0x2010  # $s0 = 0x2010
0x00002004  jr    $s0              # jump to 0x00002010
0x00002008  addi  $s1, $0, 1       # not executed
0x0000200c  sra   $s1, $s1, 2      # not executed
0x00002010  lw    $s3, 44($s1)     # executed after jr instruction
```

Code Example 6.15 shows the use of the jump register instruction (`jr`). Instruction addresses are given to the left of each instruction. `jr $s0` jumps to the address held in `$s0`, 0x00002010.

6.4.3 Conditional Statements

`if`, `if/else`, and `switch/case` statements are conditional statements commonly used in high-level languages. They each conditionally execute a *block* of code consisting of one or more statements. This section shows how to translate these high-level constructs into MIPS assembly language.

If Statements

An `if` statement executes a block of code, the *if block*, only when a condition is met. Code Example 6.16 shows how to translate an `if` statement into MIPS assembly code.

Code Example 6.16 `if` STATEMENT

High-Level Code

```
if (i == j)
  f = g + h;

f = f – i;
```

MIPS Assembly Code

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
  bne $s3, $s4, L1    # if i != j, skip if block
  add $s0, $s1, $s2   # if block: f = g + h
L1:
  sub $s0, $s0, $s3   # f = f – i
```

The assembly code for the `if` statement tests the opposite condition of the one in the high-level code. In Code Example 6.16, the high-level code tests for `i == j`, and the assembly code tests for `i != j`. The `bne` instruction branches (skips the if block) when `i != j`. Otherwise, `i == j`, the branch is not taken, and the if block is executed as desired.

If/Else Statements

`if/else` statements execute one of two blocks of code depending on a condition. When the condition in the `if` statement is met, the *if block* is executed. Otherwise, the *else block* is executed. Code Example 6.17 shows an example `if/else` statement.

Like `if` statements, `if/else` assembly code tests the opposite condition of the one in the high-level code. For example, in Code Example 6.17, the high-level code tests for `i == j`. The assembly code tests for the opposite condition (`i != j`). If that opposite condition is TRUE, `bne` skips the if block and executes the else block. Otherwise, the if block executes and finishes with a jump instruction (`j`) to jump past the else block.

Code Example 6.17 `if/else` STATEMENT

High-Level Code

```
if (i == j)
  f = g + h;

else
  f = f – i;
```

MIPS Assembly Code

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
  bne $s3, $s4, else    # if i != j, branch to else
  add $s0, $s1, $s2     # if block: f = g + h
  j   L2                # skip past the else block
else:
  sub $s0, $s0, $s3     # else block: f = f – i
L2:
```

Switch/Case Statements*

`switch/case` statements execute one of several blocks of code depending on the conditions. If no conditions are met, the `default` block is executed. A `case` statement is equivalent to a series of *nested* `if/else` statements. Code Example 6.18 shows two high-level code snippets with the same functionality: they calculate the fee for an ATM (automatic teller machine) withdrawal of $20, $50, or $100, as defined by `amount`. The MIPS assembly implementation is the same for both high-level code snippets.

6.4.4 Getting Loopy

Loops repeatedly execute a block of code depending on a condition. `for` loops and `while` loops are common loop constructs used by high-level languages. This section shows how to translate them into MIPS assembly language.

Code Example 6.18 `switch/case` STATEMENT

High-Level Code

```
switch (amount) {

  case 20:   fee = 2; break;

  case 50:   fee = 3; break;

  case 100:  fee = 5; break;

  default: fee = 0;

}

// equivalent function using if/else statements
  if      (amount == 20)  fee = 2;
  else if (amount == 50)  fee = 3;
  else if (amount == 100) fee = 5;
  else                    fee = 0;
```

MIPS Assembly Code

```
# $s0 = amount, $s1 = fee

case20:
  addi $t0, $0, 20        # $t0 = 20
  bne  $s0, $t0, case50   # amount == 20? if not,
                          # skip to case50
  addi $s1, $0, 2         # if so, fee = 2
  j    done               # and break out of case

case50:
  addi $t0, $0, 50        # $t0 = 50
  bne  $s0, $t0, case100  # amount == 50? if not,
                          # skip to case100
  addi $s1, $0, 3         # if so, fee = 3
  j    done               # and break out of case

case100:
  addi $t0, $0, 100       # $t0 = 100
  bne  $s0, $t0, default  # amount == 100? if not,
                          # skip to default
  addi $s1, $0, 5         # if so, fee = 5
  j    done               # and break out of case

default:
  add  $s1, $0, $0        # fee = 0

done:
```

While Loops

`while` loops repeatedly execute a block of code until a condition is *not* met. The `while` loop in Code Example 6.19 determines the value of `x` such that $2^x = 128$. It executes seven times, until `pow` = 128.

Like `if/else` statements, the assembly code for `while` loops tests the opposite condition of the one given in the high-level code. If that opposite condition is TRUE, the `while` loop is finished.

Code Example 6.19 `while` LOOP

High-Level Code

```
int pow = 1;
int x = 0;

while (pow != 128)
{
  pow = pow * 2;
  x = x + 1;
}
```

MIPS Assembly Code

```
# $s0 = pow, $s1 = x
  addi $s0, $0, 1       # pow = 1
  addi $s1, $0, 0       # x = 0

  addi $t0, $0, 128     # t0 = 128 for comparison
while:
  beq  $s0, $t0, done   # if pow == 128, exit while loop
  sll  $s0, $s0, 1      # pow = pow * 2
  addi $s1, $s1, 1      # x = x + 1
  j    while
done:
```

In Code Example 6.19, the `while` loop compares `pow` to 128 and exits the loop if it is equal. Otherwise it doubles `pow` (using a left shift), increments `x`, and jumps back to the start of the `while` loop.

`do/while` loops are similar to `while` loops except they execute the loop body once before checking the condition. They are of the form:

```
    do
     statement
    while (condition);
```

For Loops

`for` loops, like `while` loops, repeatedly execute a block of code until a condition is *not* met. However, `for` loops add support for a *loop variable*, which typically keeps track of the number of loop executions. A general format of the `for` loop is

```
for (initialization; condition; loop operation)
  statement
```

The `initialization` code executes before the `for` loop begins. The `condition` is tested at the beginning of each loop. If the condition is not met, the loop exits. The `loop operation` executes at the end of each loop.

Code Example 6.20 adds the numbers from 0 to 9. The loop variable, in this case `i`, is initialized to 0 and is incremented at the end of each loop iteration. At the beginning of each iteration, the `for` loop executes only when `i` is not equal to 10. Otherwise, the loop is finished. In this case, the `for` loop executes 10 times. `for` loops can be implemented using a `while` loop, but the `for` loop is often convenient.

Magnitude Comparison

So far, the examples have used `beq` and `bne` to perform equality or inequality comparisons and branches. MIPS provides the *set less than* instruction, `slt`, for magnitude comparison. `slt` sets `rd` to 1 when `rs` < `rt`. Otherwise, `rd` is 0.

Code Example 6.20 `for` LOOP

High-Level Code

```
int sum = 0;



for (i = 0; i != 10; i = i + 1) {
  sum = sum + i ;


}

// equivalent to the following while loop
int sum = 0;
int i = 0;
while (i != 10) {
  sum = sum + i;
  i = i + 1;
}
```

MIPS Assembly Code

```
# $s0 = i, $s1 = sum
  add   $s1, $0, $0      # sum = 0
  addi  $s0, $0, 0       # i = 0
  addi  $t0, $0, 10      # $t0 = 10

for:
  beq   $s0, $t0, done   # if i == 10, branch to done
  add   $s1, $s1, $s0    # sum = sum + i
  addi  $s0, $s0, 1      # increment i
  j     for

done:
```

Example 6.6 LOOPS USING `slt`

The following high-level code adds the powers of 2 from 1 to 100. Translate it into assembly language.

```
// high-level code

int sum = 0;
for (i = 1; i < 101; i = i * 2)
  sum = sum + i;
```

Solution: The assembly language code uses the set less than (`slt`) instruction to perform the less than comparison in the `for` loop.

```
# MIPS assembly code

# $s0 = i, $s1 = sum
  addi  $s1, $0, 0    # sum = 0
  addi  $s0, $0, 1    # i = 1
  addi  $t0, $0, 101  # $t0 = 101

loop:
  slt  $t1, $s0, $t0     # if (i < 101) $t1 = 1, else $t1 = 0
  beq  $t1, $0, done     # if $t1 == 0 (i >= 101), branch to done
  add  $s1, $s1, $s0     # sum = sum + i
  sll  $s0, $s0, 1       # i = i * 2
  j    loop

done:
```

Exercise 6.17 explores how to use `slt` for other magnitude comparisons including greater than, greater than or equal, and less than or equal.

6.4.5 Arrays

Arrays are useful for accessing large amounts of similar data. An array is organized as sequential data addresses in memory. Each array element is identified by a number called its *index*. The number of elements in the array is called the *size* of the array. This section shows how to access array elements in memory.

Array Indexing

Figure 6.20 shows an array of five integers stored in memory. The *index* ranges from 0 to 4. In this case, the array is stored in a processor's main memory starting at *base address* 0x10007000. The base address gives the address of the first array element, `array[0]`.

Code Example 6.21 multiplies the first two elements in `array` by 8 and stores them back into the array.

The first step in accessing an array element is to load the base address of the array into a register. Code Example 6.21 loads the base address

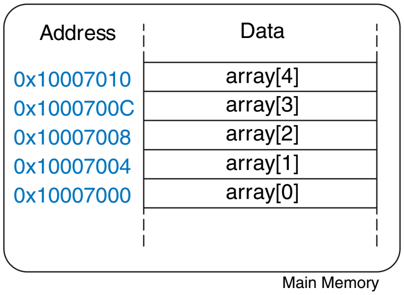


Figure 6.20 **Five-entry array with base address of 0x10007000**

Code Example 6.21 ACCESSING ARRAYS

High-Level Code

```
int array[5];


array[0] = array[0] * 8;



array[1] = array[1] * 8;
```

MIPS Assembly Code

```
# $s0 = base address of array
  lui $s0, 0x1000        # $s0 = 0x10000000
  ori $s0, $s0, 0x7000   # $s0 = 0x10007000

  lw  $t1, 0($s0)        # $t1 = array[0]
  sll $t1, $t1, 3        # $t1 = $t1 << 3 = $t1 * 8
  sw  $t1, 0($s0)        # array[0] = $t1

  Iw  $t1, 4($s0)        # $t1 = array[1]
  sll $t1, $t1, 3        # $t1 = $t1 << 3 = $t1 * 8
  sw  $t1, 4($s0)        # array[1] = $t1
```

---

into `$s0`. Recall that the load upper immediate (`lui`) and or immediate (`ori`) instructions can be used to load a 32-bit constant into a register.

Code Example 6.21 also illustrates why `lw` takes a base address and an offset. The base address points to the start of the array. The offset can be used to access subsequent elements of the array. For example, `array[1]` is stored at memory address 0x10007004 (one word or four bytes after `array[0]`), so it is accessed at an offset of 4 past the base address.

You might have noticed that the code for manipulating each of the two array elements in Code Example 6.21 is essentially the same except for the index. Duplicating the code is not a problem when accessing two array elements, but it would become terribly inefficient for accessing all of the elements in a large array. Code Example 6.22 uses a `for` loop to multiply by 8 all of the elements of a 1000-element array stored at a base address of 0x23B8F000.

Figure 6.21 shows the 1000-element array in memory. The index into the array is now a variable (`i`) rather than a constant, so we cannot take advantage of the immediate offset in `lw`. Instead, we compute the address of the *i*th element and store it in `$t0`. Remember that each array element is a word but that memory is byte addressed, so the offset from the base address is `i * 4`.

Code Example 6.22 ACCESSING ARRAYS USING A `for` LOOP

High-Level Code

```
int i;
int array[1000];

for (i = 0; i < 1000; i = i + 1)

  array[i] = array[i] * 8;
```

MIPS Assembly Code

```
# $s0 = array base address, $s1 = i
# initialization code
  lui  $s0, 0x23B8        # $s0 = 0x23B80000
  ori  $s0, $s0, 0xF000   # $s0 = 0x23B8F000
  addi $s1, $0, 0         # i = 0
  addi $t2, $0, 1000      # $t2 = 1000

loop:
  slt  $t0, $s1, $t2      # i < 1000?
  beq  $t0, $0, done      # if not, then done
  sll  $t0, $s1, 2        # $t0 = i*4 (byte offset)
  add  $t0, $t0, $s0      # address of array[i]
  lw   $t1, 0($t0)        # $t1 = array[i]
  sll  $t1, $t1, 3        # $t1 = array[i] * 8
  sw   $t1, 0($t0)        # array[i] = array[i] * 8
  addi $s1, $s1, 1        # i = i + 1
  j    loop               # repeat
done:
```

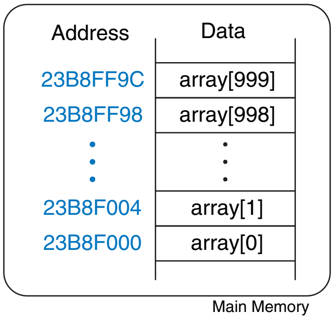


Figure 6.21 **Memory holding** `array[1000]` **starting at base address 0x23B8F000**

Shifting left by 2 is a convenient way to multiply by 4 in MIPS assembly language. This example readily extends to an array of any size.

Bytes and Characters

Numbers in the range [–128, 127] can be stored in a single byte rather than an entire word. Because there are much fewer than 256 characters on an English language keyboard, English characters are often represented by bytes. The C language uses the type `char` to represent a byte or character.

Early computers lacked a standard mapping between bytes and English characters, so exchanging text between computers was difficult. In 1963, the American Standards Association published the *American Standard Code for Information Interchange* (*ASCII*), which assigns each text character a unique byte value. Table 6.2 shows these character encodings for printable characters. The ASCII values are given in hexadecimal. Lower-case and upper-case letters differ by 0x20 (32).

Other programming languages, such as Java, use different character encodings, most notably *Unicode*. Unicode uses 16 bits to represent each character, so it supports accents, umlauts, and Asian languages. For more information, see www.unicode.org.

MIPS provides load byte and store byte instructions to manipulate bytes or characters of data: load byte unsigned (`lbu`), load byte (`lb`), and store byte (`sb`). All three are illustrated in Figure 6.22.

Table 6.2 **ASCII encodings**

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

ASCII codes developed from earlier forms of character encoding. Beginning in 1838, telegraph machines used Morse code, a series of dots (.) and dashes (–), to represent characters. For example, the letters A, B, C, and D were represented as . – , – . . . , – . – . , and – . . , respectively. The number of dots and dashes varied with each letter. For efficiency, common letters used shorter codes.

In 1874, Jean-Maurice-Emile Baudot invented a 5-bit code called the Baudot code. For example, A, B, C, and D were represented as 00011, 11001, 01110, and 01001. However, the 32 possible encodings of this 5-bit code were not sufficient for all the English characters. But 8-bit encoding was. Thus, as electronic communication became prevalent, 8-bit ASCII encoding emerged as the standard.

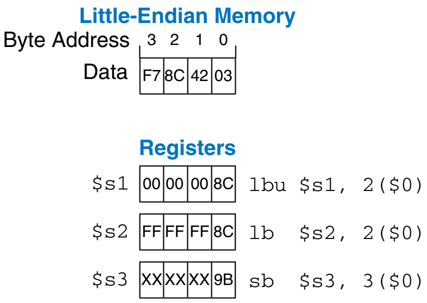


Figure 6.22 **Instructions for loading and storing bytes**

Load byte unsigned (`lbu`) zero-extends the byte, and load byte (`lb`) sign-extends the byte to fill the entire 32-bit register. Store byte (`sb`) stores the least significant byte of the 32-bit register into the specified byte address in memory. In Figure 6.22, `lbu` loads the byte at memory address `2` into the least significant byte of `$s1` and fills the remaining register bits with 0. `lb` loads the sign-extended byte at memory address `2` into `$s2`. `sb` stores the least significant byte of `$s3` into memory byte `3`; it replaces 0xF7 with 0x9B. The more significant bytes of `$s3` are ignored.

Example 6.7 USING `lb` AND `sb` TO ACCESS A CHARACTER ARRAY

The following high-level code converts a ten-entry array of characters from lower-case to upper-case by subtracting 32 from each array entry. Translate it into MIPS assembly language. Remember that the address difference between array elements is now 1 byte, not 4 bytes. Assume that `$s0` already holds the base address of `chararray`.

```
// high-level code

char chararray[10];
int i;
for (i = 0; i != 10; i = i + 1)
  chararray[i] = chararray[i] - 32;
```

Solution:

```
# MIPS assembly code
# $s0 = base address of chararray, $s1 = i

          addi $s1, $0, 0      # i = 0
          addi $t0, $0, 10     # $t0 = 10
loop:     beq  $t0, $s1, done  # if i == 10, exit loop
          add  $t1, $s1, $s0   # $t1 = address of chararray[i]
          lb   $t2, 0($t1)     # $t2 = array[i]
          addi $t2, $t2, -32   # convert to upper case: $t2 = $t2 – 32
          sb   $t2, 0($t1)     # store new value in array:
                               # chararray[i] = $t2
          addi $s1, $s1, 1     # i = i+1
          j    loop            # repeat
done:
```

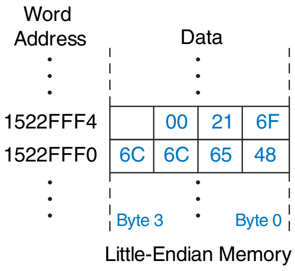


Figure 6.23 **The string "Hello!" stored in memory**

A series of characters is called a *string*. Strings have a variable length, so programming languages must provide a way to determine the length or end of the string. In C, the null character (0x00) signifies the end of a string. For example, Figure 6.23 shows the string "Hello!" (0x48 65 6C 6C 6F 21 00) stored in memory. The string is seven bytes long and extends from address 0x1522FFF0 to 0x1522FFF6. The first character of the string (H = 0x48) is stored at the lowest byte address (0x1522FFF0).

6.4.6 Function Calls

High-level languages often use *functions* (also called *procedures*) to reuse frequently accessed code and to make a program more modular and readable. Functions have inputs, called *arguments*, and an output, called the *return value*. Functions should calculate the return value and cause no other unintended side effects.

When one function calls another, the calling function, the *caller*, and the called function, the *callee*, must agree on where to put the arguments and the return value. In MIPS, the caller conventionally places up to four arguments in registers `$a0`–`$a3` before making the function call, and the callee places the return value in registers `$v0`–`$v1` before finishing. By following this convention, both functions know where to find the arguments and return value, even if the caller and callee were written by different people.

The callee must not interfere with the function of the caller. Briefly, this means that the callee must know where to return to after it completes and it must not trample on any registers or memory needed by the caller. The caller stores the *return address* in `$ra` at the same time it jumps to the callee using the jump and link instruction (`jal`). The callee must not overwrite any architectural state or memory that the caller is depending on. Specifically, the callee must leave the saved registers, `$s0`–`$s7`, `$ra`, and the *stack*, a portion of memory used for temporary variables, unmodified.

This section shows how to call and return from a function. It shows how functions access input arguments and the return value and how they use the stack to store temporary variables.

Function Calls and Returns

MIPS uses the *jump and link* instruction (`jal`) to call a function and the *jump register* instruction (`jr`) to return from a function. Code Example 6.23 shows the `main` function calling the `simple` function. `main` is the caller, and `simple` is the callee. The `simple` function is called with no input arguments and generates no return value; it simply returns to the caller. In Code Example 6.23, instruction addresses are given to the left of each MIPS instruction in hexadecimal.

Code Example 6.23 `simple` FUNCTION CALL

High-Level Code

```
int main() {
  simple();
  . . .
}
// void means the function returns no value
void simple() {
  return;
}
```

MIPS Assembly Code

```
0x00400200 main:    jal simple  # call function
0x00400204          . . .


0x00401020 simple:  jr $ra      # return
```

Jump and link (`jal`) and jump register (`jr $ra`) are the two essential instructions needed for a function call. `jal` performs two operations: it stores the address of the *next* instruction (the instruction after `jal`) in the return address register (`$ra`), and it jumps to the target instruction.

In Code Example 6.23, the `main` function calls the `simple` function by executing the jump and link (`jal`) instruction. `jal` jumps to the `simple` label and stores 0x00400204 in `$ra`. The `simple` function returns immediately by executing the instruction `jr $ra`, jumping to the instruction address held in `$ra`. The `main` function then continues executing at this address (0x00400204).

Input Arguments and Return Values

The `simple` function in Code Example 6.23 is not very useful, because it receives no input from the calling function (`main`) and returns no output. By MIPS convention, functions use `$a0`–`$a3` for input arguments and `$v0`–`$v1` for the return value. In Code Example 6.24, the function `diffofsums` is called with four arguments and returns one result.

According to MIPS convention, the calling function, `main`, places the function arguments from left to right into the input registers, `$a0`–`$a3`. The called function, `diffofsums`, stores the return value in the return register, `$v0`.

A function that returns a 64-bit value, such as a double-precision floating point number, uses both return registers, `$v0` and `$v1`. When a function with more than four arguments is called, the additional input arguments are placed on the stack, which we discuss next.

Code Example 6.24 has some subtle errors. Code Examples 6.25 and 6.26 on pages 328 and 329 show improved versions of the program.

Code Example 6.24 FUNCTION CALL WITH ARGUMENTS AND RETURN VALUES

High-Level Code

```
int main()
{
  int y;

  ...

  y = diffofsums(2, 3, 4, 5);

  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;

  result = (f + g) − (h + i);
  return result;
}
```

MIPS Assembly Code

```
# $s0 = y

main:
  ...
  addi $a0, $0, 2     # argument 0 = 2
  addi $a1, $0, 3     # argument 1 = 3
  addi $a2, $0, 4     # argument 2 = 4
  addi $a3, $0, 5     # argument 3 = 5
  jal  diffofsums     # call function
  add  $s0, $v0, $0   # y = returned value
  ...

# $s0 = result
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) − (h + i)
  add $v0, $s0, $0   # put return value in $v0
  jr  $ra            # return to caller
```

The Stack

The *stack* is memory that is used to save local variables within a function. The stack expands (uses more memory) as the processor needs more scratch space and contracts (uses less memory) when the processor no longer needs the variables stored there. Before explaining how functions use the stack to store temporary variables, we explain how the stack works.

The stack is a *last-in-first-out* (*LIFO*) *queue*. Like a stack of dishes, the last item *pushed* onto the stack (the top dish) is the first one that can be pulled (*popped*) off. Each function may allocate stack space to store local variables but must deallocate it before returning. The *top of the stack*, is the most recently allocated space. Whereas a stack of dishes grows up in space, the MIPS stack grows *down* in memory. The stack expands to lower memory addresses when a program needs more scratch space.

Figure 6.24 shows a picture of the stack. The *stack pointer*, `$sp`, is a special MIPS register that points to the top of the stack. A *pointer* is a fancy name for a memory address. It points to (gives the address of) data. For example, in Figure 6.24(a) the stack pointer, `$sp`, holds the address value 0x7FFFFFFC and points to the data value 0x12345678. `$sp` points to the top of the stack, the lowest accessible memory on the stack. Thus, in Figure 6.24(a), the stack cannot access memory below memory word 0x7FFFFFFC.

The stack pointer (`$sp`) starts at a high memory address and decrements to expand as needed. Figure 6.24(b) shows the stack expanding to allow two more data words of temporary storage. To do so, `$sp` decrements by 8 to become 0x7FFFFFF4. Two additional data words, 0xAABBCCDD and 0x11223344, are temporarily stored on the stack.

One of the important uses of the stack is to save and restore registers that are used by a function. Recall that a function should calculate a return value but have no other unintended side effects. In particular, it should not modify any registers besides the one containing the return value `$v0`. The `diffofsums` function in Code Example 6.24 violates this rule because it modifies `$t0`, `$t1`, and `$s0`. If `main` had been using `$t0`, `$t1`, or `$s0` before the call to `diffofsums`, the contents of these registers would have been corrupted by the function call.

To solve this problem, a function saves registers on the stack before it modifies them, then restores them from the stack before it returns. Specifically, it performs the following steps.

1. Makes space on the stack to store the values of one or more registers.
2. Stores the values of the registers on the stack.
3. Executes the function using the registers.
4. Restores the original values of the registers from the stack.
5. Deallocates space on the stack.

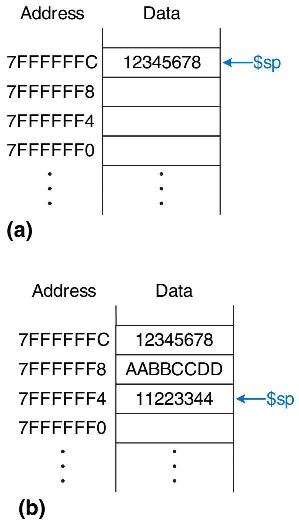


Figure 6.24 **The stack**

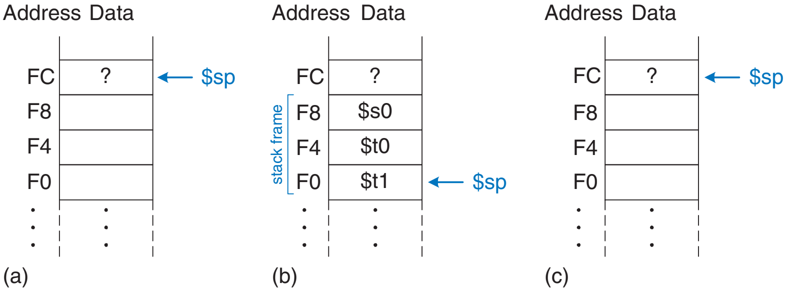


Figure 6.25 **The stack (a) before, (b) during, and (c) after** `diffofsums` **function call**

Code Example 6.25 shows an improved version of `diffofsums` that saves and restores `$t0`, `$t1`, and `$s0`. The new lines are indicated in blue. Figure 6.25 shows the stack before, during, and after a call to the `diffofsums` function from Code Example 6.25. `diffofsums` makes room for three words on the stack by decrementing the stack pointer `$sp` by 12. It then stores the current values of `$s0`, `$t0`, and `$t1` in the newly allocated space. It executes the rest of the function, changing the values in these three registers. At the end of the function, `diffofsums` restores the values of `$s0`, `$t0`, and `$t1` from the stack, deallocates its stack space, and returns. When the function returns, `$v0` holds the result, but there are no other side effects: `$s0`, `$t0`, `$t1`, and `$sp` have the same values as they did before the function call.

The stack space that a function allocates for itself is called its *stack frame*. `diffofsums`'s stack frame is three words deep. The principle of modularity tells us that each function should access only its own stack frame, not the frames belonging to other functions.

Code Example 6.25 FUNCTION SAVING REGISTERS ON THE STACK

MIPS Assembly Code

```
# $s0 = result
diffofsums:
  addi $sp, $sp, −12 # make space on stack to store three registers
  sw   $s0, 8($sp)   # save $s0 on stack
  sw   $t0, 4($sp)   # save $t0 on stack
  sw   $t1, 0($sp)   # save $t1 on stack
  add  $t0, $a0, $a1 # $t0 = f + g
  add  $t1, $a2, $a3 # $t1 = h + i
  sub  $s0, $t0, $t1 # result = (f + g) − (h + i)
  add  $v0, $s0, $0  # put return value in $v0
  lw   $t1, 0($sp)   # restore $t1 from stack
  lw   $t0, 4($sp)   # restore $t0 from stack
  lw   $s0, 8($sp)   # restore $s0 from stack
  addi $sp, $sp, 12  # deallocate stack space
  jr   $ra           # return to caller
```

Preserved Registers

Code Example 6.25 assumes that temporary registers `$t0` and `$t1` must be saved and restored. If the calling function does not use those registers, the effort to save and restore them is wasted. To avoid this waste, MIPS divides registers into *preserved* and *nonpreserved* categories. The preserved registers include `$s0`–`$s7` (hence their name, *saved*). The nonpreserved registers include `$t0`–`$t9` (hence their name, *temporary*). A function must save and restore any of the preserved registers that it wishes to use, but it can change the nonpreserved registers freely.

Code Example 6.26 shows a further improved version of `diffofsums` that saves only `$s0` on the stack. `$t0` and `$t1` are nonpreserved registers, so they need not be saved.

Remember that when one function calls another, the former is the *caller* and the latter is the *callee*. The callee must save and restore any preserved registers that it wishes to use. The callee may change any of the nonpreserved registers. Hence, if the caller is holding active data in a nonpreserved register, the caller needs to save that nonpreserved register before making the function call and then needs to restore it afterward. For these reasons, preserved registers are also called *callee-save*, and nonpreserved registers are called *caller-save*.

Code Example 6.26 FUNCTION SAVING PRESERVED REGISTERS ON THE STACK

MIPS Assembly Code

```
# $s0 = result
diffofsums
  addi  $sp, $sp, −4     # make space on stack to store one register
  sw    $s0, 0($sp)      # save $s0 on stack
  add   $t0, $a0, $a1    # $t0 = f + g
  add   $t1, $a2, $a3    # $t1 = h + i
  sub   $s0, $t0, $t1    # result = (f + g) − (h + i)
  add   $v0, $s0, $0     # put return value in $v0
  lw    $s0, 0($sp)      # restore $s0 from stack
  addi  $sp, $sp, 4      # deallocate stack space
  jr    $ra              # return to caller
```

Table 6.3 summarizes which registers are preserved. `$s0`–`$s7` are generally used to hold local variables within a function, so they must be saved. `$ra` must also be saved, so that the function knows where to return. `$t0`–`$t9` are used to hold temporary results before they are assigned to local variables. These calculations typically complete before a function call is made, so they are not preserved, and it is rare that the caller needs to save them. `$a0`–`$a3` are often overwritten in the process of calling a function.

Table 6.3 **Preserved and nonpreserved registers**

Preserved	Nonpreserved
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Return address: `$ra`	Argument registers: `$a0–$a3`
Stack pointer: `$sp`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below the stack pointer

Hence, they must be saved by the caller if the caller depends on any of its own arguments after a called function returns. `$v0–$v1` certainly should not be preserved, because the callee returns its result in these registers.

The stack above the stack pointer is automatically preserved as long as the callee does not write to memory addresses above `$sp`. In this way, it does not modify the stack frame of any other functions. The stack pointer itself is preserved, because the callee deallocates its stack frame before returning by adding back the same amount that it subtracted from `$sp` at the beginning of the function.

Recursive Function Calls

A function that does not call others is called a *leaf* function; an example is `diffofsums`. A function that does call others is called a *nonleaf* function. As mentioned earlier, nonleaf functions are somewhat more complicated because they may need to save nonpreserved registers on the stack before they call another function, and then restore those registers afterward. Specifically, the caller saves any non-preserved registers (`$t0–$t9` and `$a0–$a3`) that are needed after the call. The callee saves any of the preserved registers (`$s0–$s7` and `$ra`) that it intends to modify.

A *recursive* function is a nonleaf function that calls itself. The factorial function can be written as a recursive function call. Recall that $factorial(n) = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$. The `factorial` function can be rewritten recursively as $factorial(n) = n \times factorial(n-1)$. The factorial of 1 is simply 1. Code Example 6.27 shows the `factorial` function written as a recursive function. To conveniently refer to program addresses, we assume that the program starts at address 0x90.

The `factorial` function might modify `$a0` and `$ra`, so it saves them on the stack. It then checks whether `n` < 2. If so, it puts the return value of 1 in `$v0`, restores the stack pointer, and returns to the caller. It does not have to reload `$ra` and `$a0` in this case, because they were never modified. If `n` > 1, the function recursively calls `factorial(n-1)`. It then restores the value of `n` (`$a0`) and the return address (`$ra`) from the stack, performs the multiplication, and returns this result. The multiply

Code Example 6.27 `factorial` RECURSIVE FUNCTION CALL

High-Level Code

```
int factorial(int n) {



  if (n <= 1)
    return 1;




  else
    return (n * factorial(n − 1));
}
```

MIPS Assembly Code

```
0x90    factorial:  addi $sp, $sp, −8    # make room on stack
0x94                sw   $a0, 4($sp)     # store $a0
0x98                sw   $ra, 0($sp)     # store $ra
0x9C                addi $t0, $0, 2      # $t0 = 2
0xA0                slt  $t0, $a0, $t0   # n <= 1 ?
0xA4                beq  $t0, $0, else   # no: goto else
0xA8                addi $v0, $0, 1      # yes: return 1
0xAC                addi $sp, $sp, 8     # restore $sp
0xB0                jr   $ra             # return
0xB4         else:  addi $a0, $a0, −1    # n = n − 1
0xB8                jal  factorial       # recursive call
0xBC                Iw   $ra, 0($sp)     # restore $ra
0xC0                Iw   $a0, 4($sp)     # restore $a0
0xC4                addi $sp, $sp, 8     # restore $sp
0xC8                mul  $v0, $a0, $v0   # n * factorial(n−1)
0xCC                jr   $ra             # return
```

instruction (`mul $v0, $a0, $v0`) multiplies `$a0` and `$v0` and places the result in `$v0`.

Figure 6.26 shows the stack when executing `factorial(3)`. We assume that `$sp` initially points to 0xFC, as shown in Figure 6.26(a). The function creates a two-word stack frame to hold `$a0` and `$ra`. On the first invocation, `factorial` saves `$a0` (holding `n` = 3) at 0xF8 and `$ra` at 0xF4, as shown in Figure 6.26(b). The function then changes `$a0` to `n` = 2 and recursively calls `factorial(2)`, making `$ra` hold 0xBC. On the second invocation, it saves `$a0` (holding `n` = 2) at 0xF0 and `$ra` at 0xEC. This time, we know that `$ra` contains 0xBC. The function then changes `$a0` to `n` = 1 and recursively calls `factorial(1)`. On the third invocation, it saves `$a0` (holding `n` = 1) at 0xE8 and `$ra` at 0xE4. This time, `$ra` again contains 0xBC. The third invocation of

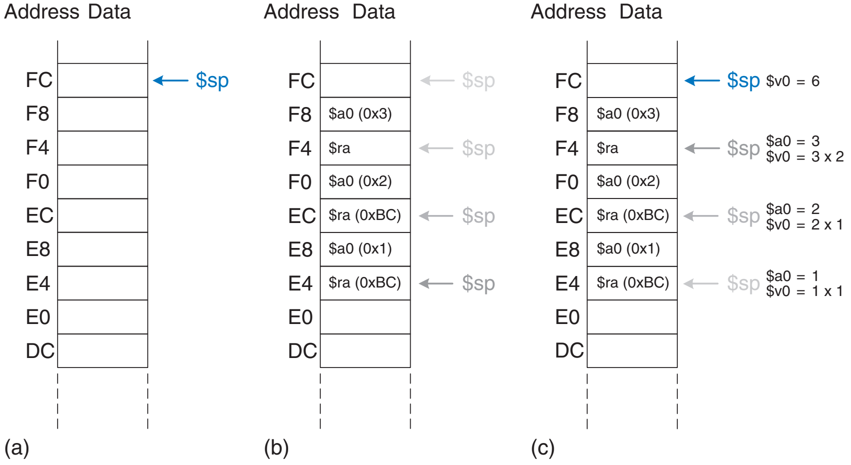

Address Data
FC
F8
F4
F0
EC
E8
E4
E0
DC
$sp
(a)
Address Data
FC
F8 $a0 (0x3)
F4 $ra
F0 $a0 (0x2)
EC $ra (0xBC)
E8 $a0 (0x1)
E4 $ra (0xBC)
E0
DC
$sp
$sp
$sp
$sp
(b)
Address Data
FC
F8 $a0 (0x3)
F4 $ra
F0 $a0 (0x2)
EC $ra (0xBC)
E8 $a0 (0x1)
E4 $ra (0xBC)
E0
DC
$sp $v0 = 6
$sp $a0 = 3 $v0 = 3 x 2
$sp $a0 = 2 $v0 = 2 x 1
$sp $a0 = 1 $v0 = 1 x 1
(c)

Figure 6.26 Stack during `factorial` function call when `n` = 3: (a) before call, (b) after last recursive call, (c) after return

`factorial` returns the value 1 in `$v0` and deallocates the stack frame before returning to the second invocation. The second invocation restores `n` to 2, restores `$ra` to 0xBC (it happened to already have this value), deallocates the stack frame, and returns `$v0` = 2 × 1 = 2 to the first invocation. The first invocation restores `n` to 3, restores `$ra` to the return address of the caller, deallocates the stack frame, and returns `$v0` = 3 × 2 = 6. Figure 6.26(c) shows the stack as the recursively called functions return. When `factorial` returns to the caller, the stack pointer is in its original position (0xFC), none of the contents of the stack above the pointer have changed, and all of the preserved registers hold their original values. `$v0` holds the return value, 6.

Additional Arguments and Local Variables*

Functions may have more than four input arguments and local variables. The stack is used to store these temporary values. By MIPS convention, if a function has more than four arguments, the first four are passed in the argument registers as usual. Additional arguments are passed on the stack, just above `$sp`. The *caller* must expand its stack to make room for the additional arguments. Figure 6.27(a) shows the caller's stack for calling a function with more than four arguments.

A function can also declare local variables or arrays. *Local* variables are declared within a function and can be accessed only within that function. Local variables are stored in `$s0`–`$s7`; if there are too many local variables, they can also be stored in the function's stack frame. In particular, local arrays are stored on the stack.

Figure 6.27(b) shows the organization of a callee's stack frame. The stack frame holds the function's own arguments, the return address, and any of the saved registers that the function will modify. It also holds local arrays and any excess local variables. If the callee has more than four

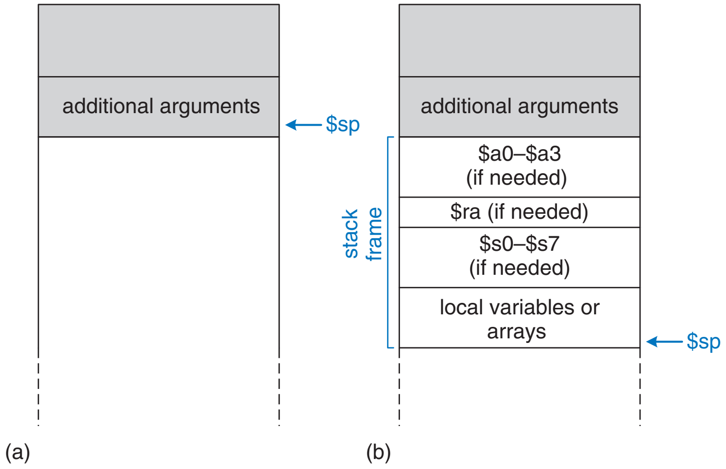


Figure 6.27 **Stack usage: (a) before call, (b) after call**

arguments, it finds them in the caller's stack frame. Accessing additional input arguments is the one exception in which a function can access stack data not in its own stack frame.

6.5 ADDRESSING MODES

MIPS uses five *addressing modes:* register-only, immediate, base, PC-relative, and pseudo-direct. The first three modes (register-only, immediate, and base addressing) define modes of reading and writing operands. The last two (PC-relative and pseudo-direct addressing) define modes of writing the program counter, `PC`.

Register-Only Addressing

Register-only addressing uses registers for all source and destination operands. All R-type instructions use register-only addressing.

Immediate Addressing

Immediate addressing uses the 16-bit immediate along with registers as operands. Some I-type instructions, such as add immediate (`addi`) and load upper immediate (`lui`), use immediate addressing.

Base Addressing

Memory access instructions, such as load word (`lw`) and store word (`sw`), use *base addressing*. The effective address of the memory operand is found by adding the base address in register `rs` to the sign-extended 16-bit offset found in the immediate field.

PC-Relative Addressing

Conditional branch instructions use *PC-relative addressing* to specify the new value of the `PC` if the branch is taken. The signed offset in the immediate field is added to the `PC` to obtain the new `PC`; hence, the branch destination address is said to be *relative* to the current `PC`.

Code Example 6.28 shows part of the `factorial` function from Code Example 6.27. Figure 6.28 shows the machine code for the `beq` instruction. The *branch target address* (*BTA*) is the address of the next instruction to execute if the branch is taken. The `beq` instruction in Figure 6.28 has a BTA of 0xB4, the instruction address of the `else` label.

The 16-bit immediate field gives the number of instructions between the BTA and the instruction *after* the branch instruction (the instruction at `PC` + 4). In this case, the value in the immediate field of `beq` is 3 because the BTA (0xB4) is 3 instructions past `PC` + 4 (0xA8).

The processor calculates the BTA from the instruction by sign-extending the 16-bit immediate, multiplying it by 4 (to convert words to bytes), and adding it to `PC` + 4.

Code Example 6.28 CALCULATING THE BRANCH TARGET ADDRESS

MIPS Assembly Code

```
0xA4          beq  $t0, $0, else
0xA8          addi $v0, $0, 1
0xAC          addi $sp, $sp, 8
0xB0          jr   $ra
0xB4    else: addi $a0, $a0, −1
0xB8          jal  factorial
```

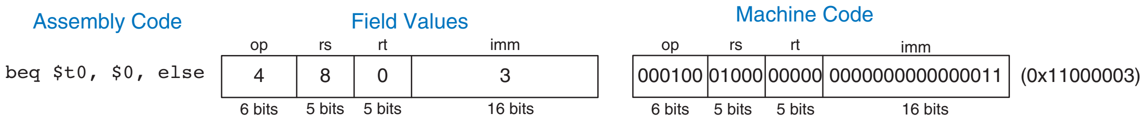

Assembly Code
beq $t0, $0, else
Field Values
op
rs
rt
imm
4
8
0
3
6 bits
5 bits
5 bits
16 bits
Machine Code
op
rs
rt
imm
000100
01000
00000
0000000000000011
(0x11000003)
6 bits
5 bits
5 bits
16 bits

Figure 6.28 **Machine code for** `beq`

Example 6.8 CALCULATING THE IMMEDIATE FIELD FOR PC-RELATIVE ADDRESSING

Calculate the immediate field and show the machine code for the branch not equal (`bne`) instruction in the following program.

```
# MIPS assembly code
0x40 loop: add  $t1, $a0, $s0
0x44       lb   $t1, 0($t1)
0x48       add  $t2, $a1, $s0
0x4C       sb   $t1, 0($t2)
0x50       addi $s0, $s0, 1
0x54       bne  $t1, $0, loop
0x58       lw   $s0, 0($sp)
```

Solution:

Figure 6.29 shows the machine code for the `bne` instruction. Its branch target address, 0x40, is 6 instructions behind `PC` + 4 (0x58), so the immediate field is –6.

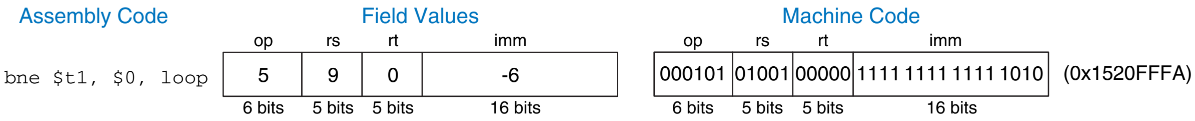

Assembly Code
bne $t1, $0, loop
Field Values
op
rs
rt
imm
5
9
0
-6
6 bits
5 bits
5 bits
16 bits
Machine Code
op
rs
rt
imm
000101
01001
00000
1111 1111 1111 1010
(0x1520FFFA)
6 bits
5 bits
5 bits
16 bits

Figure 6.29 `bne` **machine code**

Pseudo-Direct Addressing

In *direct addressing*, an address is specified in the instruction. The jump instructions, `j` and `jal`, ideally would use direct addressing to specify a

---

Code Example 6.29 CALCULATING THE JUMP TARGET ADDRESS

MIPS Assembly Code

```
0x0040005C       jal     sum
. . .
0x004000A0  sum:  add     $v0, $a0, $a1
```

32-bit *jump target address* (*JTA*) to indicate the instruction address to execute next.

Unfortunately, the J-type instruction encoding does not have enough bits to specify a full 32-bit JTA. Six bits of the instruction are used for the `opcode`, so only 26 bits are left to encode the JTA. Fortunately, the two least significant bits, JTA$_{1:0}$, should always be 0, because instructions are word aligned. The next 26 bits, JTA$_{27:2}$, are taken from the `addr` field of the instruction. The four most significant bits, JTA$_{31:28}$, are obtained from the four most significant bits of `PC` + 4. This addressing mode is called *pseudo-direct*.

Code Example 6.29 illustrates a `jal` instruction using pseudo-direct addressing. The JTA of the `jal` instruction is 0x004000A0. Figure 6.30 shows the machine code for this `jal` instruction. The top four bits and bottom two bits of the JTA are discarded. The remaining bits are stored in the 26-bit address field (`addr`).

The processor calculates the JTA from the J-type instruction by appending two 0's and prepending the four most significant bits of `PC` + 4 to the 26-bit address field (`addr`).

Because the four most significant bits of the JTA are taken from `PC` + 4, the jump range is limited. The range limits of branch and jump instructions are explored in Exercises 6.29 to 6.32. All J-type instructions, `j` and `jal`, use pseudo-direct addressing.

Note that the jump register instruction, `jr`, is *not* a J-type instruction. It is an R-type instruction that jumps to the 32-bit value held in register `rs`.

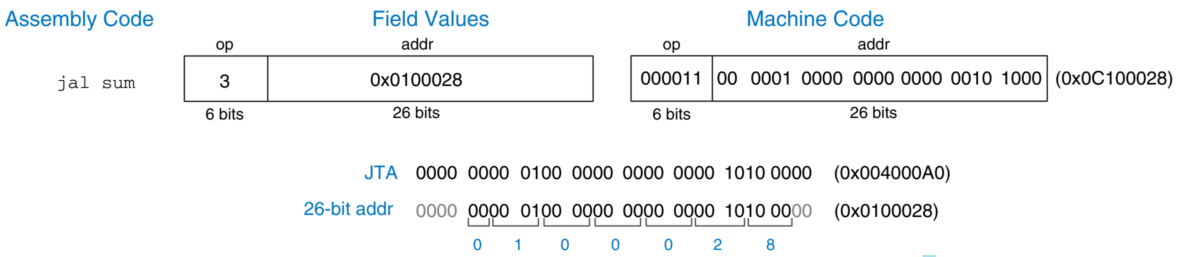

Assembly Code
Field Values
Machine Code
jal sum
op
addr
3
0x0100028
6 bits
26 bits
op
addr
000011
00 0001 0000 0000 0000 0010 1000
(0x0C100028)
6 bits
26 bits
JTA
0000 0000 0100 0000 0000 0000 1010 0000
(0x004000A0)
26-bit addr
0000 0000 0100 0000 0000 0000 1010 0000
(0x0100028)
0 1 0 0 0 2 8

Figure 6.30 `jal` **machine code**

6.6 LIGHTS, CAMERA, ACTION: COMPILING, ASSEMBLING, AND LOADING

Up until now, we have shown how to translate short high-level code snippets into assembly and machine code. This section describes how to compile and assemble a complete high-level program and how to load the program into memory for execution.

We begin by introducing the MIPS *memory map*, which defines where code, data, and stack memory are located. We then show the steps of code execution for a sample program.

6.6.1 The Memory Map

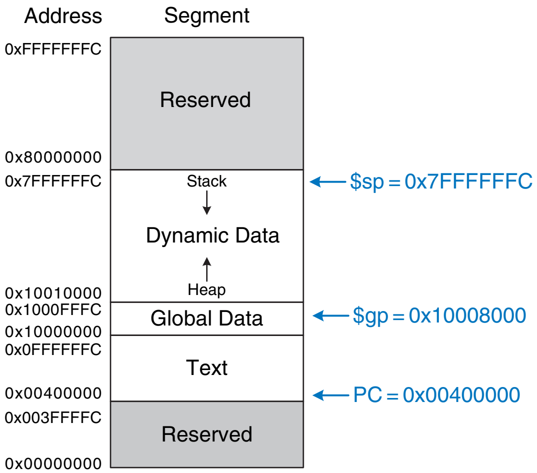
With 32-bit addresses, the MIPS *address space* spans 2^{32} bytes = 4 gigabytes (GB). Word addresses are divisible by 4 and range from 0 to 0xFFFFFFFC. Figure 6.31 shows the MIPS memory map. The MIPS architecture divides the address space into four parts or *segments:* the text segment, global data segment, dynamic data segment, and reserved segments. The following sections describe each segment.

The Text Segment

The *text segment* stores the machine language program. It is large enough to accommodate almost 256 MB of code. Note that the four most significant bits of the address in the text space are all 0, so the `j` instruction can directly jump to any address in the program.

The Global Data Segment

The *global data segment* stores global variables that, in contrast to local variables, can be seen by all functions in a program. Global variables



Address
Segment
0xFFFFFFFC
Reserved
0x80000000
0x7FFFFFFC
Stack
Dynamic Data
Heap
0x10010000
0x1000FFFC
Global Data
0x10000000
0x0FFFFFFC
Text
0x00400000
0x003FFFFC
Reserved
0x00000000
$sp = 0x7FFFFFFC
$gp = 0x10008000
PC = 0x00400000

Figure 6.31 **MIPS memory map**

are defined at *start-up*, before the program begins executing. These variables are declared outside the main function in a C program and can be accessed by any function. The global data segment is large enough to store 64 KB of global variables.

Global variables are accessed using the global pointer (`$gp`), which is initialized to 0x100080000. Unlike the stack pointer (`$sp`), `$gp` does not change during program execution. Any global variable can be accessed with a 16-bit positive or negative offset from `$gp`. The offset is known at assembly time, so the variables can be efficiently accessed using base addressing mode with constant offsets.

The Dynamic Data Segment

The *dynamic data segment* holds the stack and the *heap*. The data in this segment are not known at start-up but are dynamically allocated and deallocated throughout the execution of the program. This is the largest segment of memory used by a program, spanning almost 2 GB of the address space.

As discussed in Section 6.4.6, the stack is used to save and restore registers used by functions and to hold local variables such as arrays. The stack grows downward from the top of the dynamic data segment (0x7FFFFFFC) and each stack frame is accessed in last-in-first-out order.

The heap stores data that is allocated by the program during runtime. In C, memory allocations are made by the `malloc` function; in C++ and Java, `new` is used to allocate memory. Like a heap of clothes on a dorm room floor, heap data can be used and discarded in any order. The heap grows upward from the bottom of the dynamic data segment.

If the stack and heap ever grow into each other, the program's data can become corrupted. The memory allocator tries to ensure that this never happens by returning an out-of-memory error if there is insufficient space to allocate more dynamic data.

The Reserved Segments

The *reserved segments* are used by the operating system and cannot directly be used by the program. Part of the reserved memory is used for interrupts (see Section 7.7) and for memory-mapped I/O (see Section 8.5).

Grace Hopper, 1906–1992. Graduated from Yale University with a Ph.D. in mathematics. Developed the first compiler while working for the Remington Rand Corporation and was instrumental in developing the COBOL programming language. As a naval officer, she received many awards, including a World War II Victory Medal and the National Defense Service Medal.

6.6.2 Translating and Starting a Program

Figure 6.32 shows the steps required to translate a program from a high-level language into machine language and to start executing that program. First, the high-level code is compiled into assembly code. The assembly code is assembled into machine code in an *object file*. The linker combines the machine code with object code from libraries and other files to produce an entire executable program. In practice, most compilers perform all three steps of compiling, assembling, and linking. Finally, the loader

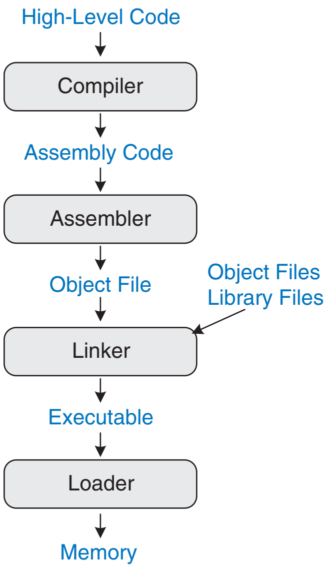

High-Level Code
Compiler
Assembly Code
Assembler
Object File
Object Files
Library Files
Linker
Executable
Loader
Memory

Figure 6.32 Steps for translating and starting a program

loads the program into memory and starts execution. The remainder of this section walks through these steps for a simple program.

Step 1: Compilation

A compiler translates high-level code into assembly language. Code Example 6.30 shows a simple high-level program with three global variables and

---

Code Example 6.30 COMPILING A HIGH-LEVEL PROGRAM

High-Level Code

```
int f, g, y; // global variables




int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}






int sum(int a, int b) {
  return (a + b);
}
```

MIPS Assembly Code

```
.data
f:
g:
y:

.text
main:
  addi $sp, $sp, −4  # make stack frame
  sw   $ra, 0($sp)   # store $ra on stack
  addi $a0, $0, 2    # $a0 = 2
  sw   $a0, f        # f = 2
  addi $a1, $0, 3    # $a1 = 3
  sw   $a1, g        # g = 3
  jal  sum           # call sum function
  sw   $v0, y        # y = sum(f, g)
  lw   $ra, 0($sp)   # restore $ra from stack
  addi $sp, $sp, 4   # restore stack pointer
  jr   $ra           # return to operating system

sum:
  add  $v0, $a0, $a1 # $v0 = a + b
  jr   $ra           # return to caller
```

two functions, along with the assembly code produced by a typical compiler. The `.data` and `.text` keywords are *assembler directives* that indicate where the text and data segments begin. Labels are used for global variables `f`, `g`, and `y`. Their storage location will be determined by the assembler; for now, they are left as symbols in the code.

Step 2: Assembling

The assembler turns the assembly language code into an *object file* containing machine language code. The assembler makes two passes through the assembly code. On the first pass, the assembler assigns instruction addresses and finds all the *symbols*, such as labels and global variable names. The code after the first assembler pass is shown here.

```
0x00400000 main: addi $sp, $sp, −4
0x00400004       sw   $ra, 0($sp)
0x00400008       addi $a0, $0, 2
0x0040000C       sw   $a0, f
0x00400010       addi $a1, $0, 3
0x00400014       sw   $a1, g
0x00400018       jal  sum
0x0040001C       sw   $v0, y
0x00400020       lw   $ra, 0($sp)
0x00400024       addi $sp, $sp, 4
0x00400028       jr   $ra
0x0040002C sum:  add  $v0, $a0, $a1
0x00400030       jr   $ra
```

The names and addresses of the symbols are kept in a *symbol table*, as shown in Table 6.4 for this code. The symbol addresses are filled in after the first pass, when the addresses of labels are known. Global variables are assigned storage locations in the global data segment of memory, starting at memory address 0x10000000.

On the second pass through the code, the assembler produces the machine language code. Addresses for the global variables and labels are taken from the symbol table. The machine language code and symbol table are stored in the object file.

Table 6.4 Symbol table

Symbol	Address
`f`	0x10000000
`g`	0x10000004
`y`	0x10000008
`main`	0x00400000
`sum`	0x0040002C

Step 3: Linking

Most large programs contain more than one file. If the programmer changes only one of the files, it would be wasteful to recompile and reassemble the other files. In particular, programs often call functions in library files; these library files almost never change. If a file of high-level code is not changed, the associated object file need not be updated.

The job of the linker is to combine all of the object files into one machine language file called the *executable*. The linker relocates the data and instructions in the object files so that they are not all on top of each other. It uses the information in the symbol tables to adjust the addresses of global variables and of labels that are relocated.

In our example, there is only one object file, so no relocation is necessary. Figure 6.33 shows the executable file. It has three sections: the executable file header, the text segment, and the data segment. The executable file header reports the text size (code size) and data size (amount of globally declared data). Both are given in units of bytes. The text segment gives the instructions in the order that they are stored in memory.

The figure shows the instructions in human-readable format next to the machine code for ease of interpretation, but the executable file includes only machine instructions. The data segment gives the address of each global variable. The global variables are addressed with respect to the base address given by the global pointer, `$gp`. For example, the first

Executable file header	Text Size	Data Size	
	0x34 (52 bytes)	0xC (12 bytes)	
Text segment	**Address**	**Instruction**	
	0x00400000	0x23BDFFFC	addi $sp, $sp, –4
	0x00400004	0xAFBF0000	sw $ra, 0($sp)
	0x00400008	0x20040002	addi $a0, $0, 2
	0x0040000C	0xAF848000	sw $a0, 0x8000($gp)
	0x00400010	0x20050003	addi $a1, $0, 3
	0x00400014	0xAF858004	sw $a1, 0x8004($gp)
	0x00400018	0x0C10000B	jal 0x0040002C
	0x0040001C	0xAF828008	sw $v0, 0x8008($gp)
	0x00400020	0x8FBF0000	lw $ra, 0($sp)
	0x00400024	0x23BD0004	addi $sp, $sp, –4
	0x00400028	0x03E00008	jr $ra
	0x0040002C	0x00851020	add $v0, $a0, $a1
	0x00400030	0x03E00008	jr $ra
Data segment	**Address**	**Data**	
	0x10000000	f	
	0x10000004	g	
	0x10000008	y	

Figure 6.33 Executable

store instruction, `sw $a0, 0x8000($gp)`, stores the value 2 to the global variable `f`, which is located at memory address 0x10000000. Remember that the offset, 0x8000, is a 16-bit signed number that is sign-extended and added to the base address, `$gp`. So, `$gp` + 0x8000 = 0x10008000 + 0xFFFF8000 = 0x10000000, the memory address of variable `f`.

Step 4: Loading

The operating system loads a program by reading the text segment of the executable file from a storage device (usually the hard disk) into the text segment of memory. The operating system sets `$gp` to 0x10008000 (the middle of the global data segment) and `$sp` to 0x7FFFFFFC (the top of the dynamic data segment), then performs a `jal 0x00400000` to jump to the beginning of the program. Figure 6.34 shows the memory map at the beginning of program execution.

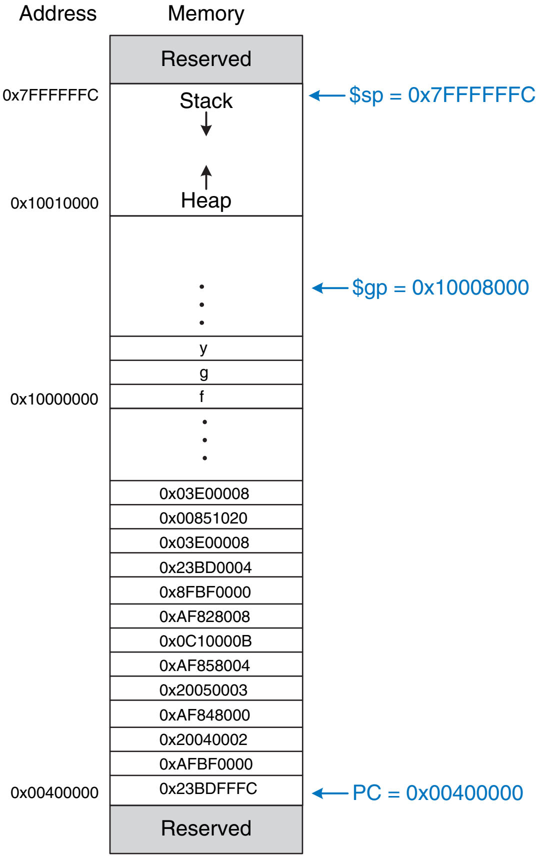

Address
Memory
Reserved
0x7FFFFFFC
Stack
$sp = 0x7FFFFFFC
0x10010000
Heap
$gp = 0x10008000
y
g
0x10000000
f
0x03E00008
0x00851020
0x03E00008
0x23BD0004
0x8FBF0000
0xAF828008
0x0C10000B
0xAF858004
0x20050003
0xAF848000
0x20040002
0xAFBF0000
0x00400000
0x23BDFFFC
PC = 0x00400000
Reserved

Figure 6.34 Executable loaded in memory

6.7 ODDS AND ENDS*

This section covers a few optional topics that do not fit naturally elsewhere in the chapter. These topics include pseudoinstructions, exceptions, signed and unsigned arithmetic instructions, and floating-point instructions.

6.7.1 Pseudoinstructions

If an instruction is not available in the MIPS instruction set, it is probably because the same operation can be performed using one or more existing MIPS instructions. Remember that MIPS is a reduced instruction set computer (RISC), so the instruction size and hardware complexity are minimized by keeping the number of instructions small.

However, MIPS defines *pseudoinstructions* that are not actually part of the instruction set but are commonly used by programmers and compilers. When converted to machine code, pseudoinstructions are translated into one or more MIPS instructions.

Table 6.5 gives examples of pseudoinstructions and the MIPS instructions used to implement them. For example, the load immediate pseudoinstruction (`li`) loads a 32-bit constant using a combination of `lui` and `ori` instructions. The no operation pseudoinstruction (`nop`, pronounced "no op") performs no operation. The `PC` is incremented by 4 upon its execution. No other registers or memory values are altered. The machine code for the `nop` instruction is 0x00000000.

Some pseudoinstructions require a temporary register for intermediate calculations. For example, the pseudoinstruction `beq $t2,` $\texttt{imm}_{15:0}$`, Loop` compares `$t2` to a 16-bit immediate, $\texttt{imm}_{15:0}$. This pseudoinstruction requires a temporary register in which to store the 16-bit immediate. Assemblers use the assembler register, `$at`, for such purposes. Table 6.6 shows

Table 6.5 Pseudoinstructions

Pseudoinstruction	Corresponding MIPS Instructions
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`
`clear $t0`	`add $t0, $0, $0`
`move $s2, $s1`	`add $s2, $s1, $0`
`nop`	`sll $0, $0, 0`

Table 6.6 **Pseudoinstruction using** `$at`

Pseudoinstruction	Corresponding MIPS Instructions
`beq $t2,` $\text{imm}_{15:0}$`, Loop`	`addi $at, $0,` $\text{imm}_{15:0}$ `beq  $t2, $at, Loop`

how the assembler uses `$at` in converting a pseudoinstruction to real MIPS instructions. We leave it as Exercises 6.38 and 6.39 to implement other pseudoinstructions such as rotate left (`rol`) and rotate right (`ror`).

6.7.2 Exceptions

An *exception* is like an unscheduled function call that jumps to a new address. Exceptions may be caused by hardware or software. For example, the processor may receive notification that the user pressed a key on a keyboard. The processor may stop what it is doing, determine which key was pressed, save it for future reference, then resume the program that was running. Such a hardware exception triggered by an input/output (I/O) device such as a keyboard is often called an *interrupt*. Alternatively, the program may encounter an error condition such as an undefined instruction. The program then jumps to code in the *operating system* (*OS*), which may choose to terminate the offending program. Software exceptions are sometimes called *traps*. Other causes of exceptions include division by zero, attempts to read nonexistent memory, hardware malfunctions, debugger breakpoints, and arithmetic overflow (see Section 6.7.3).

The processor records the cause of an exception and the value of the `PC` at the time the exception occurs. It then jumps to the *exception handler* function. The exception handler is code (usually in the OS) that examines the cause of the exception and responds appropriately (by reading the keyboard on a hardware interrupt, for example). It then returns to the program that was executing before the exception took place. In MIPS, the exception handler is always located at 0x80000180. When an exception occurs, the processor always jumps to this instruction address, regardless of the cause.

The MIPS architecture uses a special-purpose register, called the `Cause` register, to record the cause of the exception. Different codes are used to record different exception causes, as given in Table 6.7. The exception handler code reads the `Cause` register to determine how to handle the exception. Some other architectures jump to a different exception handler for each different cause instead of using a `Cause` register.

MIPS uses another special-purpose register called the Exception Program Counter (`EPC`) to store the value of the `PC` at the time an

Table 6.7 Exception cause codes

Exception	Cause
hardware interrupt	`0x00000000`
system call	`0x00000020`
breakpoint/divide by 0	`0x00000024`
undefined instruction	`0x00000028`
arithmetic overflow	`0x00000030`

exception takes place. The processor returns to the address in `EPC` after handling the exception. This is analogous to using `$ra` to store the old value of the `PC` during a `jal` instruction.

The `EPC` and `Cause` registers are not part of the MIPS register file. The `mfc0` (move from coprocessor 0) instruction copies these and other special-purpose registers into one of the general purpose registers. Coprocessor 0 is called the *MIPS processor control;* it handles interrupts and processor diagnostics. For example, `mfc0 $t0, Cause` copies the `Cause` register into `$t0`.

The `syscall` and `break` instructions cause traps to perform system calls or debugger breakpoints. The exception handler uses the `EPC` to look up the instruction and determine the nature of the system call or breakpoint by looking at the fields of the instruction.

In summary, an exception causes the processor to jump to the exception handler. The exception handler saves registers on the stack, then uses `mfc0` to look at the cause and respond accordingly. When the handler is finished, it restores the registers from the stack, copies the return address from `EPC` to `$k0` using `mfc0`, and returns using `jr $k0`.

`$k0` and `$k1` are included in the MIPS register set. They are reserved by the OS for exception handling. They do not need to be saved and restored during exceptions.

6.7.3 Signed and Unsigned Instructions

Recall that a binary number may be signed or unsigned. The MIPS architecture uses two's complement representation of signed numbers. MIPS has certain instructions that come in signed and unsigned flavors, including addition and subtraction, multiplication and division, set less than, and partial word loads.

Addition and Subtraction

Addition and subtraction are performed identically whether the number is signed or unsigned. However, the interpretation of the results is different.

As mentioned in Section 1.4.6, if two large signed numbers are added together, the result may incorrectly produce the opposite sign. For example, adding the following two huge positive numbers gives a negative result: 0x7FFFFFFF + 0x7FFFFFFF = 0xFFFFFFFE = −2. Similarly, adding two huge negative numbers gives a positive result, 0x80000001 + 0x80000001 = 0x00000002. This is called arithmetic *overflow*.

The C language ignores arithmetic overflows, but other languages, such as Fortran, require that the program be notified. As mentioned in Section 6.7.2, the MIPS processor takes an exception on arithmetic overflow. The program can decide what to do about the overflow (for example, it might repeat the calculation with greater precision to avoid the overflow), then return to where it left off.

MIPS provides signed and unsigned versions of addition and subtraction. The signed versions are `add`, `addi`, and `sub`. The unsigned versions are `addu`, `addiu`, and `subu`. The two versions are identical except that signed versions trigger an exception on overflow, whereas unsigned versions do not. Because C ignores exceptions, C programs technically use the unsigned versions of these instructions.

Multiplication and Division

Multiplication and division behave differently for signed and unsigned numbers. For example, as an unsigned number, 0xFFFFFFFF represents a large number, but as a signed number it represents –1. Hence, 0xFFFFFFFF × 0xFFFFFFFF would equal 0xFFFFFFFE00000001 if the numbers were unsigned but 0x0000000000000001 if the numbers were signed.

Therefore, multiplication and division come in both signed and unsigned flavors. `mult` and `div` treat the operands as signed numbers. `multu` and `divu` treat the operands as unsigned numbers.

Set Less Than

Set less than instructions can compare either two registers (`slt`) or a register and an immediate (`slti`). Set less than also comes in signed (`slt` and `slti`) and unsigned (`sltu` and `sltiu`) versions. In a signed comparison, 0x80000000 is less than any other number, because it is the most negative two's complement number. In an unsigned comparison, 0x80000000 is greater than 0x7FFFFFFF but less than 0x80000001, because all numbers are positive.

Beware that `sltiu` sign-extends the immediate before treating it as an unsigned number. For example, `sltiu $s0, $s1, 0x8042` compares `$s1` to 0xFFFF8042, treating the immediate as a large positive number.

Loads

As described in Section 6.4.5, byte loads come in signed (`lb`) and unsigned (`lbu`) versions. `lb` sign-extends the byte, and `lbu` zero-extends the byte to fill the entire 32-bit register. Similarly, MIPS provides signed and unsigned half-word loads (`lh` and `lhu`), which load two bytes into the lower half and sign- or zero-extend the upper half of the word.

6.7.4 Floating-Point Instructions

The MIPS architecture defines an optional floating-point coprocessor, known as coprocessor 1. In early MIPS implementations, the floating-point coprocessor was a separate chip that users could purchase if they needed fast floating-point math. In most recent MIPS implementations, the floating-point coprocessor is built in alongside the main processor.

MIPS defines thirty-two 32-bit floating-point registers, `$f0`–`$f31`. These are separate from the ordinary registers used so far. MIPS supports both single- and double-precision IEEE floating-point arithmetic. Double-precision (64-bit) numbers are stored in pairs of 32-bit registers, so only the 16 even-numbered registers (`$f0`, `$f2`, `$f4`, . . . , `$f30`) are used to specify double-precision operations. By convention, certain registers are reserved for certain purposes, as given in Table 6.8.

Floating-point instructions all have an `opcode` of 17 (10001_2). They require both a `funct` field and a `cop` (coprocessor) field to indicate the type of instruction. Hence, MIPS defines the *F-type* instruction format for floating-point instructions, shown in Figure 6.35. Floating-point instructions come in both single- and double-precision flavors. `cop` = 16 (10000_2) for single-precision instructions or 17 (10001_2) for double-precision instructions. Like R-type instructions, F-type instructions have two source operands, `fs` and `ft`, and one destination, `fd`.

Instruction precision is indicated by `.s` and `.d` in the mnemonic. Floating-point arithmetic instructions include addition (`add.s`, `add.d`), subtraction (`sub.s`, `sub.d`), multiplication (`mul.s`, `mul.d`), and division (`div.s`, `div.d`) as well as negation (`neg.s`, `neg.d`) and absolute value (`abs.s`, `abs.d`).

Table 6.8 **MIPS floating-point register set**

Name	Number	Use
`$fv0`–`$fv1`	0, 2	function return value
`$ft0`–`$ft3`	4, 6, 8, 10	temporary variables
`$fa0`–`$fa1`	12, 14	function arguments
`$ft4`–`$ft5`	16, 18	temporary variables
`$fs0`–`$fs5`	20, 22, 24, 26, 28, 30	saved variables

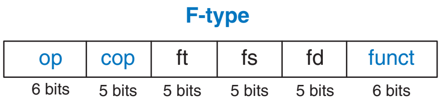


Figure 6.35 **F-type machine instruction format**

Floating-point branches have two parts. First, a compare instruction is used to set or clear the *floating-point condition flag* (`fpcond`). Then, a conditional branch checks the value of the flag. The compare instructions include equality (`c.seq.s`/`c.seq.d`), less than (`c.lt.s`/`c.lt.d`), and less than or equal to (`c.le.s`/`c.le.d`). The conditional branch instructions are `bc1f` and `bc1t` that branch if `fpcond` is FALSE or TRUE, respectively. Inequality, greater than or equal to, and greater than comparisons are performed with `seq`, `lt`, and `le`, followed by `bc1f`.

Floating-point registers are loaded and stored from memory using `lwc1` and `swc1`. These instructions move 32 bits, so two are necessary to handle a double-precision number.

6.8 REAL-WORLD PERSPECTIVE: x86 ARCHITECTURE*

Almost all personal computers today use x86 architecture microprocessors. x86, also called IA-32, is a 32-bit architecture originally developed by Intel. AMD also sells x86 compatible microprocessors.

The x86 architecture has a long and convoluted history dating back to 1978, when Intel announced the 16-bit 8086 microprocessor. IBM selected the 8086 and its cousin, the 8088, for IBM's first personal computers. In 1985, Intel introduced the 32-bit 80386 microprocessor, which was backward compatible with the 8086, so it could run software developed for earlier PCs. Processor architectures compatible with the 80386 are called x86 processors. The Pentium, Core, and Athlon processors are well known x86 processors. Section 7.9 describes the evolution of x86 microprocessors in more detail.

Various groups at Intel and AMD over many years have shoehorned more instructions and capabilities into the antiquated architecture. The result is far less elegant than MIPS. As Patterson and Hennessy explain, "this checkered ancestry has led to an architecture that is difficult to explain and impossible to love." However, software compatibility is far more important than technical elegance, so x86 has been the *de facto* PC standard for more than two decades. More than 100 million x86 processors are sold every year. This huge market justifies more than $5 billion of research and development annually to continue improving the processors.

x86 is an example of a Complex Instruction Set Computer (CISC) architecture. In contrast to RISC architectures such as MIPS, each CISC instruction can do more work. Programs for CISC architectures usually require fewer instructions. The instruction encodings were selected to be more compact, so as to save memory, when RAM was far more expensive than it is today; instructions are of variable length and are often less than 32 bits. The trade-off is that complicated instructions are more difficult to decode and tend to execute more slowly.

Table 6.9 Major differences between MIPS and x86

Feature	MIPS	x86
# of registers	32 general purpose	8, some restrictions on purpose
# of operands	3 (2 source, 1 destination)	2 (1 source, 1 source/destination)
operand location	registers or immediates	registers, immediates, or memory
operand size	32 bits	8, 16, or 32 bits
condition codes	no	yes
instruction types	simple	simple and complicated
instruction encoding	fixed, 4 bytes	variable, 1–15 bytes

This section introduces the x86 architecture. The goal is not to make you into an x86 assembly language programmer, but rather to illustrate some of the similarities and differences between x86 and MIPS. We think it is interesting to see how x86 works. However, none of the material in this section is needed to understand the rest of the book. Major differences between x86 and MIPS are summarized in Table 6.9.

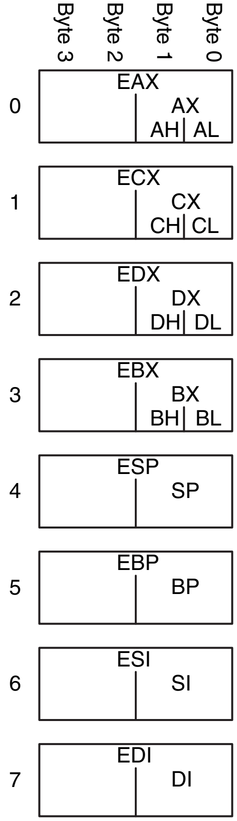


Figure 6.36 x86 registers

6.8.1 x86 Registers

The 8086 microprocessor provided eight 16-bit registers. It could separately access the upper and lower eight bits of some of these registers. When the 32-bit 80386 was introduced, the registers were extended to 32 bits. These registers are called `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI`, and `EDI`. For backward compatibility, the bottom 16 bits and some of the bottom 8-bit portions are also usable, as shown in Figure 6.36.

The eight registers are almost, but not quite, general purpose. Certain instructions cannot use certain registers. Other instructions always put their results in certain registers. Like `$sp` in MIPS, `ESP` is normally reserved for the stack pointer.

The x86 program counter is called `EIP` (the *extended instruction pointer*). Like the MIPS `PC`, it advances from one instruction to the next or can be changed with branch, jump, and function call instructions.

6.8.2 x86 Operands

MIPS instructions always act on registers or immediates. Explicit load and store instructions are needed to move data between memory and the registers. In contrast, x86 instructions may operate on registers, immediates, or memory. This partially compensates for the small set of registers.

MIPS instructions generally specify three operands: two sources and one destination. x86 instructions specify only two operands. The first is a source. The second is both a source and the destination. Hence, x86 instructions always overwrite one of their sources with the result. Table 6.10 lists the combinations of operand locations in x86. All combinations are possible except memory to memory.

Like MIPS, x86 has a 32-bit memory space that is byte-addressable. However, x86 also supports a much wider variety of memory *addressing modes*. Memory locations are specified with any combination of a *base register*, *displacement*, and a *scaled index register*. Table 6.11 illustrates these combinations. The displacement can be an 8-, 16-, or 32-bit value. The scale multiplying the index register can be 1, 2, 4, or 8. The base + displacement mode is equivalent to the MIPS base addressing mode for loads and stores. The scaled index provides an easy way to access arrays or structures of 2-, 4-, or 8-byte elements without having to issue a sequence of instructions to generate the address.

While MIPS always acts on 32-bit words, x86 instructions can operate on 8-, 16-, or 32-bit data. Table 6.12 illustrates these variations.

Table 6.10 **Operand locations**

Source/ Destination	Source	Example	Meaning
register	register	`add EAX, EBX`	`EAX <- EAX + EBX`
register	immediate	`add EAX, 42`	`EAX <- EAX + 42`
register	memory	`add EAX, [20]`	`EAX <- EAX + Mem[20]`
memory	register	`add [20], EAX`	`Mem[20] <- Mem[20] + EAX`
memory	immediate	`add [20], 42`	`Mem[20] <- Mem[20] + 42`

Table 6.11 **Memory addressing modes**

Example	Meaning	Comment
`add EAX, [20]`	`EAX <- EAX + Mem[20]`	displacement
`add EAX, [ESP]`	`EAX <- EAX + Mem[ESP]`	base addressing
`add EAX, [EDX+40]`	`EAX <- EAX + Mem[EDX+40]`	base + displacement
`add EAX, [60+EDI*4]`	`EAX <- EAX + Mem[60+EDI*4]`	displacement + scaled index
`add EAX, [EDX+80+EDI*2]`	`EAX <- EAX + Mem[EDX+80+EDI*2]`	base + displacement + scaled index

Table 6.12 Instructions acting on 8-, 16-, or 32-bit data

Example	Meaning	Data Size
`add AH, BL`	`AH <- AH + BL`	8-bit
`add AX, −1`	`AX <- AX + 0xFFFF`	16-bit
`add EAX, EDX`	`EAX <- EAX + EDX`	32-bit

6.8.3 Status Flags

x86, like many CISC architectures, uses *status flags* (also called *condition codes*) to make decisions about branches and to keep track of carries and arithmetic overflow. x86 uses a 32-bit register, called `EFLAGS`, that stores the status flags. Some of the bits of the `EFLAGS` register are given in Table 6.13. Other bits are used by the operating system.

The architectural state of an x86 processor includes `EFLAGS` as well as the eight registers and the `EIP`.

6.8.4 x86 Instructions

x86 has a larger set of instructions than MIPS. Table 6.14 describes some of the general purpose instructions. x86 also has instructions for floating-point arithmetic and for arithmetic on multiple short data elements packed into a longer word. `D` indicates the destination (a register or memory location), and `S` indicates the source (a register, memory location, or immediate).

Note that some instructions always act on specific registers. For example, 32 × 32-bit multiplication always takes one of the sources from `EAX` and always puts the 64-bit result in `EDX` and `EAX`. `LOOP` always stores the loop counter in `ECX`. `PUSH`, `POP`, `CALL`, and `RET` use the stack pointer, `ESP`.

Conditional jumps check the flags and branch if the appropriate condition is met. They come in many flavors. For example, `JZ` jumps if the

Table 6.13 Selected EFLAGS

Name	Meaning
`CF` (Carry Flag)	Carry out generated by last arithmetic operation. Indicates overflow in unsigned arithmetic. Also used for propagating the carry between words in multiple-precision arithmetic.
`ZF` (Zero Flag)	Result of last operation was zero.
`SF` (Sign Flag)	Result of last operation was negative (msb = 1).
`OF` (Overflow Flag)	Overflow of two's complement arithmetic.

Table 6.14 Selected x86 instructions

Instruction	Meaning	Function
`ADD/SUB`	add/subtract	`D = D + S / D = D – S`
`ADDC`	add with carry	`D = D + S + CF`
`INC/DEC`	increment/decrement	`D = D + 1 / D = D – 1`
`CMP`	compare	Set flags based on `D – S`
`NEG`	negate	`D = –D`
`AND/OR/XOR`	logical AND/OR/XOR	`D = D op S`
`NOT`	logical NOT	$D = \overline{D}$
`IMUL/MUL`	signed/unsigned multiply	`EDX:EAX = EAX × D`
`IDIV/DIV`	signed/unsigned divide	`EDX:EAX/D` `EAX` = Quotient; `EDX` = Remainder
`SAR/SHR`	arithmetic/logical shift right	`D = D >>> S / D = D >> S`
`SAL/SHL`	left shift	`D = D << S`
`ROR/ROL`	rotate right/left	Rotate `D` by `S`
`RCR/RCL`	rotate right/left with carry	Rotate `CF` and `D` by `S`
`BT`	bit test	`CF = D[S]` (the `S`*th* bit of `D`)
`BTR/BTS`	bit test and reset/set	`CF = D[S]; D[S] = 0 / 1`
`TEST`	set flags based on masked bits	Set flags based on `D AND S`
`MOV`	move	`D = S`
`PUSH`	push onto stack	`ESP = ESP – 4; Mem[ESP] = S`
`POP`	pop off stack	`D = MEM[ESP]; ESP = ESP + 4`
`CLC, STC`	clear/set carry flag	`CF = 0 / 1`
`JMP`	unconditional jump	relative jump: `EIP = EIP + S` absolute jump: `EIP = S`
`Jcc`	conditional jump	`if (flag) EIP = EIP + S`
`LOOP`	loop	`ECX = ECX – 1` `if (ECX ≠ 0) EIP = EIP + imm`
`CALL`	function call	`ESP = ESP - 4;` `MEM[ESP] = EIP; EIP = S`
`RET`	function return	`EIP = MEM[ESP]; ESP = ESP + 4`

Table 6.15 **Selected branch conditions**

Instruction	Meaning	Function After `CMP D, S`
`JZ/JE`	jump if `ZF` = 1	jump if `D` = `S`
`JNZ/JNE`	jump if `ZF` = 0	jump if `D` ≠ `S`
`JGE`	jump if `SF` = `OF`	jump if `D` ≥ `S`
`JG`	jump if `SF` = `OF` and `ZF` = 0	jump if `D` > `S`
`JLE`	jump if `SF` ≠ `OF` or `ZF` = 1	jump if `D` ≤ `S`
`JL`	jump if `SF` ≠ `OF`	jump if `D` < `S`
`JC/JB`	jump if `CF` = 1	
`JNC`	jump if `CF` = 0	
`JO`	jump if `OF` = 1	
`JNO`	jump if `OF` = 0	
`JS`	jump if `SF` = 1	
`JNS`	jump if `SF` = 0	

zero flag (`ZF`) is 1. `JNZ` jumps if the zero flag is 0. The jumps usually follow an instruction, such as the compare instruction (`CMP`), that sets the flags. Table 6.15 lists some of the conditional jumps and how they depend on the flags set by a prior compare operation.

6.8.5 x86 Instruction Encoding

The x86 instruction encodings are truly messy, a legacy of decades of piecemeal changes. Unlike MIPS, whose instructions are uniformly 32 bits, x86 instructions vary from 1 to 15 bytes, as shown in Figure 6.37.[3] The `opcode` may be 1, 2, or 3 bytes. It is followed by four optional fields: `Mod R/M`, `SIB`, `Displacement`, and `Immediate`. `ModR/M` specifies an addressing mode. `SIB` specifies the scale, index, and base registers in certain addressing modes. `Displacement` indicates a 1-, 2-, or 4-byte displacement in certain addressing modes. And `Immediate` is a 1-, 2-, or 4-byte constant for instructions using an immediate as the source operand. Moreover, an

---

[3] It is possible to construct 17-byte instructions if all the optional fields are used. However, x86 places a 15-byte limit on the length of legal instructions.

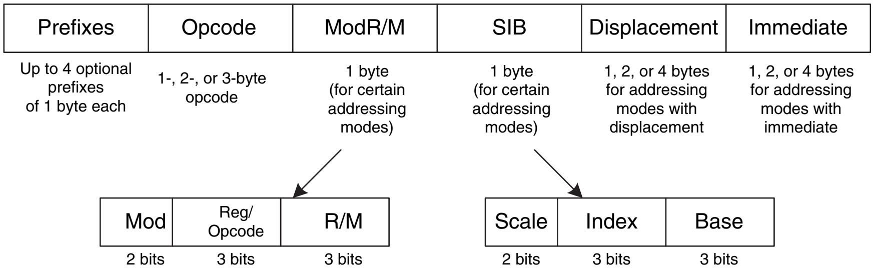

Prefixes
Opcode
ModR/M
SIB
Displacement
Immediate
Up to 4 optional prefixes of 1 byte each
1-, 2-, or 3-byte opcode
1 byte (for certain addressing modes)
1 byte (for certain addressing modes)
1, 2, or 4 bytes for addressing modes with displacement
1, 2, or 4 bytes for addressing modes with immediate
Mod
Reg/ Opcode
R/M
2 bits
3 bits
3 bits
Scale
Index
Base
2 bits
3 bits
3 bits

Figure 6.37 **x86 instruction encodings**

instruction can be preceded by up to four optional byte-long prefixes that modify its behavior.

The `ModR/M` byte uses the 2-bit `Mod` and 3-bit `R/M` field to specify the addressing mode for one of the operands. The operand can come from one of the eight registers, or from one of 24 memory addressing modes. Due to artifacts in the encodings, the `ESP` and `EBP` registers are not available for use as the base or index register in certain addressing modes. The `Reg` field specifies the register used as the other operand. For certain instructions that do not require a second operand, the `Reg` field is used to specify three more bits of the `opcode`.

In addressing modes using a scaled index register, the `SIB` byte specifies the index register and the scale (1, 2, 4, or 8). If both a base and index are used, the `SIB` byte also specifies the base register.

MIPS fully specifies the instruction in the `opcode` and `funct` fields of the instruction. x86 uses a variable number of bits to specify different instructions. It uses fewer bits to specify more common instructions, decreasing the average length of the instructions. Some instructions even have multiple opcodes. For example, `add AL, imm8` performs an 8-bit add of an immediate to `AL`. It is represented with the 1-byte opcode, 0x04, followed by a 1-byte immediate. The `A` register (`AL`, `AX`, or `EAX`) is called the *accumulator*. On the other hand, `add D, imm8` performs an 8-bit add of an immediate to an arbitrary destination, `D` (memory or a register). It is represented with the 1-byte `opcode` 0x80 followed by one or more bytes specifying `D`, followed by a 1-byte immediate. Many instructions have shortened encodings when the destination is the accumulator.

In the original 8086, the `opcode` specified whether the instruction acted on 8- or 16-bit operands. When the 80386 introduced 32-bit operands, no new opcodes were available to specify the 32-bit form. Instead, the same opcode was used for both 16- and 32-bit forms. An additional bit in the *code segment descriptor* used by the OS specifies which form the processor should choose. The bit is set to 0 for backward compatibility with 8086 programs, defaulting the `opcode` to 16-bit operands. It is set to 1 for programs to default to 32-bit operands. Moreover, the programmer can specify prefixes to change the form for a particular instruction. If the `prefix` 0x66 appears before the `opcode`, the alternative size operand is used (16 bits in 32-bit mode, or 32 bits in 16-bit mode).

6.8.6 Other x86 Peculiarities

The 80286 introduced *segmentation* to divide memory into segments of up to 64 KB in length. When the OS enables segmentation, addresses are computed relative to the beginning of the segment. The processor checks for addresses that go beyond the end of the segment and indicates an error, thus preventing programs from accessing memory outside their own segment. Segmentation proved to be a hassle for programmers and is not used in modern versions of the Windows operating system.

x86 contains string instructions that act on entire strings of bytes or words. The operations include moving, comparing, or scanning for a specific value. In modern processors, these instructions are usually slower than performing the equivalent operation with a series of simpler instructions, so they are best avoided.

As mentioned earlier, the 0x66 `prefix` is used to choose between 16- and 32-bit operand sizes. Other prefixes include ones used to lock the bus (to control access to shared variables in a multiprocessor system), to predict whether a branch will be taken or not, and to repeat the instruction during a string move.

Intel and Hewlett-Packard jointly developed a new 64-bit architecture called IA-64 in the mid 1990's. It was designed from a clean slate, bypassing the convoluted history of x86, taking advantage of 20 years of new research in computer architecture, and providing a 64-bit address space. However, IA-64 has yet to become a market success. Most computers needing the large address space now use the 64-bit extensions of x86.

The bane of any architecture is to run out of memory capacity. With 32-bit addresses, x86 can access 4 GB of memory. This was far more than the largest computers had in 1985, but by the early 2000s it had become limiting. In 2003, AMD extended the address space and register sizes to 64 bits, calling the enhanced architecture AMD64. AMD64 has a compatibility mode that allows it to run 32-bit programs unmodified while the OS takes advantage of the bigger address space. In 2004, Intel gave in and adopted the 64-bit extensions, renaming them Extended Memory 64 Technology (EM64T). With 64-bit addresses, computers can access 16 exabytes (16 billion GB) of memory.

For those curious about more details of the x86 architecture, the x86 Intel Architecture Software Developer's Manual is freely available on Intel's Web site.

6.8.7 The Big Picture

This section has given a taste of some of the differences between the MIPS RISC architecture and the x86 CISC architecture. x86 tends to have shorter programs, because a complex instruction is equivalent to a series of simple MIPS instructions and because the instructions are encoded to minimize memory use. However, the x86 architecture is a hodgepodge of features accumulated over the years, some of which are no longer useful but must be kept for compatibility with old programs. It has too few registers, and the instructions are difficult to decode. Merely explaining the instruction set is difficult. Despite all these failings, x86 is firmly entrenched as the dominant computer architecture for PCs, because the value of software compatibility is so great and because the huge market justifies the effort required to build fast x86 microprocessors.

6.9 SUMMARY

To command a computer, you must speak its language. A computer architecture defines how to command a processor. Many different computer architectures are in widespread commercial use today, but once you understand one, learning others is much easier. The key questions to ask when approaching a new architecture are

- What is the data word length?
- What are the registers?
- How is memory organized?
- What are the instructions?

MIPS is a 32-bit architecture because it operates on 32-bit data. The MIPS architecture has 32 general-purpose registers. In principle, almost any register can be used for any purpose. However, by convention, certain registers are reserved for certain purposes, for ease of programming and so that functions written by different programmers can communicate easily. For example, register 0 (`$0`) always holds the constant 0, `$ra` holds the return address after a `jal` instruction, and `$a0`–`$a3` and `$v0`–`$v1` hold the arguments and return value of a function. MIPS has a byte-addressable memory system with 32-bit addresses. The memory map was described in Section 6.6.1. Instructions are 32 bits long and must be word aligned. This chapter discussed the most commonly used MIPS instructions.

The power of defining a computer architecture is that a program written for any given architecture can run on many different implementations of that architecture. For example, programs written for the Intel Pentium processor in 1993 will generally still run (and run much faster) on the Intel Xeon or AMD Phenom processors in 2012.

In the first part of this book, we learned about the circuit and logic levels of abstraction. In this chapter, we jumped up to the architecture level. In the next chapter, we study microarchitecture, the arrangement of digital building blocks that implement a processor architecture. Microarchitecture is the link between hardware and software engineering. And, we believe it is one of the most exciting topics in all of engineering: you will learn to build your own microprocessor!

Exercises

Exercise 6.1 Give three examples from the MIPS architecture of each of the architecture design principles: (1) simplicity favors regularity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises. Explain how each of your examples exhibits the design principle.

Exercise 6.2 The MIPS architecture has a register set that consists of 32-bit registers. Is it possible to design a computer architecture without a register set? If so, briefly describe the architecture, including the instruction set. What are advantages and disadvantages of this architecture over the MIPS architecture?

Exercise 6.3 Consider memory storage of a 32-bit word stored at memory word 42 in a byte-addressable memory.

(a) What is the byte address of memory word 42?

(b) What are the byte addresses that memory word 42 spans?

(c) Draw the number 0xFF223344 stored at word 42 in both big-endian and little-endian machines. Your drawing should be similar to Figure 6.4. Clearly label the byte address corresponding to each data byte value.

Exercise 6.4 Repeat Exercise 6.3 for memory storage of a 32-bit word stored at memory word 15 in a byte-addressable memory.

Exercise 6.5 Explain how the following program can be used to determine whether a computer is big-endian or little-endian:

```
li $t0, 0xABCD9876
sw $t0, 100($0)
lb $s5, 101($0)
```

Exercise 6.6 Write the following strings using ASCII encoding. Write your final answers in hexadecimal.

(a) SOS

(b) Cool!

(c) (your own name)

Exercise 6.7 Repeat Exercise 6.6 for the following strings.

(a) howdy

(b) lions

(c) To the rescue!

Exercise 6.8 Show how the strings in Exercise 6.6 are stored in a byte-addressable memory on (a) a big-endian machine and (b) a little-endian machine starting at memory address 0x1000100C. Use a memory diagram similar to Figure 6.4. Clearly indicate the memory address of each byte on each machine.

Exercise 6.9 Repeat Exercise 6.8 for the strings in Exercise 6.7.

Exercise 6.10 Convert the following MIPS assembly code into machine language. Write the instructions in hexadecimal.

```
add  $t0, $s0, $s1
lw   $t0, 0x20($t7)
addi $s0, $0, −10
```

Exercise 6.11 Repeat Exercise 6.10 for the following MIPS assembly code:

```
addi $s0, $0, 73
sw   $t1, −7($t2)
sub  $t1, $s7, $s2
```

Exercise 6.12 Consider I-type instructions.

(a) Which instructions from Exercise 6.10 are I-type instructions?

(b) Sign-extend the 16-bit immediate of each instruction from part (a) so that it becomes a 32-bit number.

Exercise 6.13 Repeat Exercise 6.12 for the instructions in Exercise 6.11.

Exercise 6.14 Convert the following program from machine language into MIPS assembly language. The numbers on the left are the instruction addresses in memory, and the numbers on the right give the instruction at that address. Then reverse engineer a high-level program that would compile into this assembly language routine and write it. Explain in words what the program does. `$a0` is the input, and it initially contains a positive number, `n`. `$v0` is the output.

```
0x00400000  0x20080000
0x00400004  0x20090001
0x00400008  0x0089502A
0x0040000C  0x15400003
0x00400010  0x01094020
0x00400014  0x21290002
0x00400018  0x08100002
0x0040001C  0x01001020
0x00400020  0x03E00008
```

Exercise 6.15 Repeat Exercise 6.14 for the following machine code. `$a0` and `$a1` are the inputs. `$a0` contains a 32-bit number and `$a1` is the address of a 32-element array of characters (`char`).

```
0x00400000  0x2008001F
0x00400004  0x01044806
0x00400008  0x31290001
0x0040000C  0x0009482A
0x00400010  0xA0A90000
0x00400014  0x20A50001
0x00400018  0x2108FFFF
0x0040001C  0x0501FFF9
0x00400020  0x03E00008
```

Exercise 6.16 The `nori` instruction is not part of the MIPS instruction set, because the same functionality can be implemented using existing instructions. Write a short assembly code snippet that has the following functionality: `$t0 = $t1 NOR 0xF234`. Use as few instructions as possible.

Exercise 6.17 Implement the following high-level code segments using the `slt` instruction. Assume the integer variables `g` and `h` are in registers `$s0` and `$s1`, respectively.

(a)
```
if (g > h)
  g = g + h;
else
  g = g − h;
```

(b)
```
if (g >= h)
  g = g + 1;
else
  h = h − 1;
```

(c)
```
if (g <= h)
  g = 0;
else
  h = 0;
```

Exercise 6.18 Write a function in a high-level language for `int find42(int array[], int size)`. `size` specifies the number of elements in `array`, and `array` specifies the base address of the array. The function should return the index number of the first array entry that holds the value 42. If no array entry is 42, it should return the value –1.

Exercise 6.19 The high-level function `strcpy` copies the character string `src` to the character string `dst` (see page 360).

This simple string copy function has a serious flaw: it has no way of knowing that `dst` has enough space to receive `src`. If a malicious programmer were able to execute `strcpy` with a long string `src`, the programmer might be able to write bytes all over memory, possibly even modifying code stored in subsequent memory locations. With some cleverness, the modified code might take over the machine. This is called a buffer overflow attack; it is employed by several nasty programs, including the infamous Blaster worm, which caused an estimated $525 million in damages in 2003.

```
// C code
void strcpy(char dst[], char src[]) {
  int i = 0;

  do {
    dst[i] = src[i];
  } while (src[i++]);
}
```

(a) Implement the `strcpy` function in MIPS assembly code. Use `$s0` for `i`.

(b) Draw a picture of the stack before, during, and after the `strcpy` function call. Assume `$sp` = 0x7FFFFF00 just before `strcpy` is called.

Exercise 6.20 Convert the high-level function from Exercise 6.18 into MIPS assembly code.

Exercise 6.21 Consider the MIPS assembly code below. `func1`, `func2`, and `func3` are non-leaf functions. `func4` is a leaf function. The code is not shown for each function, but the comments indicate which registers are used within each function.

```
0x00401000    func1 : . . .        # func1 uses $s0–$s1
0x00401020            jal func2
 . . .
0x00401100    func2:  . . .        # func2 uses $s2–$s7
0x0040117C            jal func3
 . . .
0x00401400    func3:  . . .        # func3 uses $s1–$s3
0x00401704            jal func4
 . . .
0x00403008    func4:  . . .        # func4 uses no preserved
0x00403118            jr $ra       # registers
```

(a) How many words are the stack frames of each function?

(b) Sketch the stack after `func4` is called. Clearly indicate which registers are stored where on the stack and mark each of the stack frames. Give values where possible.

Exercise 6.22 Each number in the *Fibonacci series* is the sum of the previous two numbers. Table 6.16 lists the first few numbers in the series, *fib*(*n*).

Table 6.16 Fibonacci series

n	1	2	3	4	5	6	7	8	9	10	11	. . .
$fib(n)$	1	1	2	3	5	8	13	21	34	55	89	. . .

(a) What is *fib*(*n*) for $n = 0$ and $n = -1$?

(b) Write a function called `fib` in a high-level language that returns the Fibonacci number for any nonnegative value of *n*. Hint: You probably will want to use a loop. Clearly comment your code.

(c) Convert the high-level function of part (b) into MIPS assembly code. Add comments after every line of code that explain clearly what it does. Use the SPIM simulator to test your code on *fib*(9). (See the Preface for how to install the SPIM simulator.)

Exercise 6.23 Consider C Code Example 6.27. For this exercise, assume `factorial` is called with input argument `n` = 5.

(a) What value is in `$v0` when `factorial` returns to the calling function?

(b) Suppose you delete the instructions at addresses 0x98 and 0xBC that save and restore `$ra`. Will the program (1) enter an infinite loop but not crash; (2) crash (cause the stack to grow beyond the dynamic data segment or the PC to jump to a location outside the program); (3) produce an incorrect value in `$v0` when the program returns to loop (if so, what value?), or (4) run correctly despite the deleted lines?

(c) Repeat part (b) when the instructions at the following instruction addresses are deleted:

  (i) 0x94 and 0xC0 (instructions that save and restore `$a0`)
  (ii) 0x90 and 0xC4 (instructions that save and restore `$sp`). Note: the `factorial` label is not deleted
  (iii) 0xAC (an instruction that restores `$sp`)

Exercise 6.24 Ben Bitdiddle is trying to compute the function $f(a, b) = 2a + 3b$ for nonnegative *b*. He goes overboard in the use of function calls and recursion and produces the following high-level code for functions `f` and `f2`.

```
// high-level code for functions f and f2
int f(int a, int b) {
  int j;
  j = a;
  return j + a + f2(b);
}

int f2(int x)
{
  int k;
  k = 3;
  if (x == 0) return 0;
  else return k + f2(x − 1);
}
```

Ben then translates the two functions into assembly language as follows. He also writes a function, `test`, that calls the function `f(5, 3)`.

```
# MIPS assembly code
# f: $a0 = a, $a1 = b, $s0 = j; f2: $a0 = x, $s0 = k

0x00400000  test: addi $a0, $0, 5     # $a0 = 5 (a = 5)
0x00400004        addi $a1, $0, 3     # $a1 = 3 (b = 3)
0x00400008        jal  f              # call f(5, 3)
0x0040000C  loop: j    loop           # and loop forever

0x00400010  f:    addi $sp, $sp, −16  # make room on the stack
                                      # for $s0, $a0, $a1, and $ra
0x00400014        sw   $a1, 12($sp)   # save $a1 (b)
0x00400018        sw   $a0, 8($sp)    # save $a0 (a)
0x0040001C        sw   $ra, 4($sp)    # save $ra
0x00400020        sw   $s0, 0($sp)    # save $s0
0x00400024        add  $s0, $a0, $0   # $s0 = $a0 (j = a)
0x00400028        add  $a0, $a1, $0   # place b as argument for f2
0x0040002C        jal  f2             # call f2(b)
0x00400030        lw   $a0, 8($sp)    # restore $a0 (a) after call
0x00400034        lw   $a1, 12($sp)   # restore $a1 (b) after call
0x00400038        add  $v0, $v0, $s0  # $v0 = f2(b) + j
0x0040003C        add  $v0, $v0, $a0  # $v0 = (f2(b) + j) + a
0x00400040        lw   $s0, 0($sp)    # restore $s0
0x00400044        lw   $ra, 4($sp)    # restore $ra
0x00400048        addi $sp, $sp, 16   # restore $sp (stack pointer)
0x0040004C        jr   $ra            # return to point of call

0x00400050  f2:   addi $sp, $sp, −12  # make room on the stack for
                                      # $s0, $a0, and $ra
0x00400054        sw   $a0, 8($sp)    # save $a0 (x)
0x00400058        sw   $ra, 4($sp)    # save return address
0x0040005C        sw   $s0, 0($sp)    # save $s0
0x00400060        addi $s0, $0, 3     # k = 3
0x00400064        bne  $a0, $0, else  # x = 0?
0x00400068        addi $v0, $0, 0     # yes: return value should be 0
0x0040006C        j    done           # and clean up
0x00400070  else: addi $a0, $a0, −1   # no: $a0 = $a0 − 1 (x = x − 1)
0x00400074        jal  f2             # call f2(x − 1)
0x00400078        lw   $a0, 8($sp)    #  restore $a0 (x)
0x0040007C        add  $v0, $v0, $s0  #  $v0 = f2(x − 1) + k
0x00400080  done: lw   $s0, 0($sp)    #  restore $s0
0x00400084        lw   $ra, 4($sp)    #  restore $ra
0x00400088        addi $sp, $sp, 12   #  restore $sp
0x0040008C        jr   $ra            #  return to point of call
```

You will probably find it useful to make drawings of the stack similar to the one in Figure 6.26 to help you answer the following questions.

(a) If the code runs starting at `test`, what value is in `$v0` when the program gets to `loop`? Does his program correctly compute *$2a + 3b$*?

(b) Suppose Ben deletes the instructions at addresses 0x0040001C and 0x00400044 that save and restore `$ra`. Will the program (1) enter an infinite loop but not crash; (2) crash (cause the stack to grow beyond the dynamic data segment or the `PC` to jump to a location outside the program); (3) produce an incorrect value in `$v0` when the program returns to loop (if so, what value?), or (4) run correctly despite the deleted lines?

(c) Repeat part (b) when the instructions at the following instruction addresses are deleted. Note that labels aren't deleted, only instructions.

(i) 0x00400018 and 0x00400030 (instructions that save and restore `$a0`)

(ii) 0x00400014 and 0x00400034 (instructions that save and restore `$a1`)

(iii) 0x00400020 and 0x00400040 (instructions that save and restore `$s0`)

(iv) 0x00400050 and 0x00400088 (instructions that save and restore `$sp`)

(v) 0x0040005C and 0x00400080 (instructions that save and restore `$s0`)

(vi) 0x00400058 and 0x00400084 (instructions that save and restore `$ra`)

(vii) 0x00400054 and 0x00400078 (instructions that save and restore `$a0`)

Exercise 6.25 Convert the following `beq`, `j`, and `jal` assembly instructions into machine code. Instruction addresses are given to the left of each instruction.

(a)

```
0x00401000          beq $t0, $s1, Loop
0x00401004           . . .
0x00401008           . . .
0x0040100C   Loop:   . . .
```

(b)

```
0x00401000           beq $t7, $s4, done
 . . .               . . .
0x00402040   done:   . . .
```

(c)

```
0x0040310C   back:   . . .
 . . .               . . .
0x00405000           beq $t9, $s7, back
```

(d)

```
0x00403000           jal func
 . . .               . . .
0x0041147C   func:   . . .
```

(e)

```
0x00403004   back:   . . .
 . . .               . . .
0x0040400C   j       back
```

Exercise 6.26 Consider the following MIPS assembly language snippet. The numbers to the left of each instruction indicate the instruction address.

```
0x00400028       add  $a0, $a1, $0
0x0040002C       jal  f2
0x00400030 f1:   jr   $ra
0x00400034 f2:   sw   $s0, 0($s2)
0x00400038       bne  $a0, $0, else
0x0040003C       j    f1
0x00400040 else: addi $a0, $a0, −1
0x00400044       j    f2
```

(a) Translate the instruction sequence into machine code. Write the machine code instructions in hexadecimal.

(b) List the addressing mode used at each line of code.

Exercise 6.27 Consider the following C code snippet.

```
//  C code
void setArray(int num) {
  int i;
  int array[10];

  for (i = 0; i < 10; i = i + 1) {
    array[i] = compare(num, i);
  }
}

int compare(int a, int b) {
  if (sub(a, b) >= 0)
    return 1;
  else
  return 0;
}

int sub(int a, int b) {
  return a − b;
}
```

(a) Implement the C code snippet in MIPS assembly language. Use `$s0` to hold the variable `i`. Be sure to handle the stack pointer appropriately. The array is stored on the stack of the `setArray` function (see Section 6.4.6).

(b) Assume `setArray` is the first function called. Draw the status of the stack before calling `setArray` and during each function call. Indicate the names of registers and variables stored on the stack, mark the location of `$sp`, and clearly mark each stack frame.

(c) How would your code function if you failed to store `$ra` on the stack?

Exercise 6.28 Consider the following high-level function.

```
// C code
int f(int n, int k) {
  int b;

  b = k + 2;
  if (n == 0) b = 10;
  else b = b + (n * n) + f(n − 1, k + 1);
  return b * k;
}
```

(a) Translate the high-level function `f` into MIPS assembly language. Pay particular attention to properly saving and restoring registers across function calls and using the MIPS preserved register conventions. Clearly comment your code. You can use the MIPS `mul` instruction. The function starts at instruction address 0x00400100. Keep local variable `b` in `$s0`.

(b) Step through your function from part (a) by hand for the case of `f(2, 4)`. Draw a picture of the stack similar to the one in Figure 6.26(c). Write the register name and data value stored at each location in the stack and keep track of the stack pointer value (`$sp`). Clearly mark each stack frame. You might also find it useful to keep track of the values in `$a0`, `$a1`, `$v0`, and `$s0` throughout execution. Assume that when `f` is called, `$s0` = 0xABCD and `$ra` = 0x400004. What is the final value of `$v0`?

Exercise 6.29 What is the range of instruction addresses to which conditional branches, such as `beq` and `bne`, can branch in MIPS? Give your answer in number of instructions relative to the conditional branch instruction.

Exercise 6.30 The following questions examine the limitations of the jump instruction, `j`. Give your answer in number of instructions relative to the jump instruction.

(a) In the worst case, how far can the jump instruction (`j`) jump forward (i.e., to higher addresses)? (The worst case is when the jump instruction cannot jump far.) Explain using words and examples, as needed.

(b) In the best case, how far can the jump instruction (`j`) jump forward? (The best case is when the jump instruction can jump the farthest.) Explain.

(c) In the worst case, how far can the jump instruction (`j`) jump backward (to lower addresses)? Explain.

(d) In the best case, how far can the jump instruction (`j`) jump backward? Explain.

Exercise 6.31 Explain why it is advantageous to have a large address field, `addr`, in the machine format for the jump instructions, `j` and `jal`.

Exercise 6.32 Write assembly code that jumps to an instruction 64 Minstructions from the first instruction. Recall that 1 Minstruction = 2^{20} instructions = 1,048,576 instructions. Assume that your code begins at address 0x00400000. Use a minimum number of instructions.

Exercise 6.33 Write a function in high-level code that takes a 10-entry array of 32-bit integers stored in little-endian format and converts it to big-endian format. After writing the high-level code, convert it to MIPS assembly code. Comment all your code and use a minimum number of instructions.

Exercise 6.34 Consider two strings: `string1` and `string2`.

(a) Write high-level code for a function called concat that concatenates (joins together) the two strings: `void concat(char string1[], char string2[], char stringconcat[])`. The function does not return a value. It concatenates `string1` and `string2` and places the resulting string in `stringconcat`. You may assume that the character array `stringconcat` is large enough to accommodate the concatenated string.

(b) Convert the function from part (a) into MIPS assembly language.

Exercise 6.35 Write a MIPS assembly program that adds two positive single-precision floating point numbers held in `$s0` and `$s1`. Do not use any of the MIPS floating-point instructions. You need not worry about any of the encodings that are reserved for special purposes (e.g., 0, NANs, etc.) or numbers that overflow or underflow. Use the SPIM simulator to test your code. You will need to manually set the values of `$s0` and `$s1` to test your code. Demonstrate that your code functions reliably.

Exercise 6.36 Show how the following MIPS program would be loaded into memory and executed.

```
# MIPS assembly code
main:
   addi $sp, $sp, −4
   sw $ra, 0($sp)
   lw $a0, x
   lw $a1, y
   jal diff
   lw $ra, 0($sp)
   addi $sp, $sp, 4
   jr $ra
diff:
   sub $v0, $a0, $a1
   jr $ra
```

(a) First show the instruction address next to each assembly instruction.

(b) Draw the symbol table showing the labels and their addresses.

(c) Convert all instructions into machine code.

(d) How big (how many bytes) are the data and text segments?

(e) Sketch a memory map showing where data and instructions are stored.

Exercise 6.37 Repeat Exercise 6.36 for the following MIPS code.

```
# MIPS assembly code
main:
   addi $sp, $sp, −4
   sw $ra, 0($sp)
   addi $t0, $0, 15
   sw $t0, a
   addi $a1, $0, 27
   sw $a1, b
   lw $a0, a
   jal greater
   lw $ra, 0($sp)
   addi $sp, $sp, 4
   jr $ra
greater:
   slt $v0, $a1, $a0
   jr $ra
```

Exercise 6.38 Show the MIPS instructions that implement the following pseudoinstructions. You may use the assembler register, `$at`, but you may not corrupt (overwrite) any other registers.

(a) `addi $t0, $s2,` $imm_{31:0}$

(b) `lw $t5,` $imm_{31:0}$`($s0)`

(c) `rol $t0, $t1, 5` (rotate `$t1` left by 5 and put the result in `$t0`)

(d) `ror $s4, $t6, 31` (rotate `$t6` right by 31 and put the result in `$s4`)

Exercise 6.39 Repeat Exercise 6.38 for the following pseudoinstructions.

(a) `beq  $t1, imm`$_{31:0}$`, L`

(b) `ble  $t3, $t5, L`

(c) `bgt  $t3, $t5, L`

(d) `bge  $t3, $t5, L`

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs (but are usually open to any assembly language).

Question 6.1 Write MIPS assembly code for swapping the contents of two registers, `$t0` and `$t1`. You may not use any other registers.

Question 6.2 Suppose you are given an array of both positive and negative integers. Write MIPS assembly code that finds the subset of the array with the largest sum. Assume that the array's base address and the number of array elements are in `$a0` and `$a1`, respectively. Your code should place the resulting subset of the array starting at base address `$a2`. Write code that runs as fast as possible.

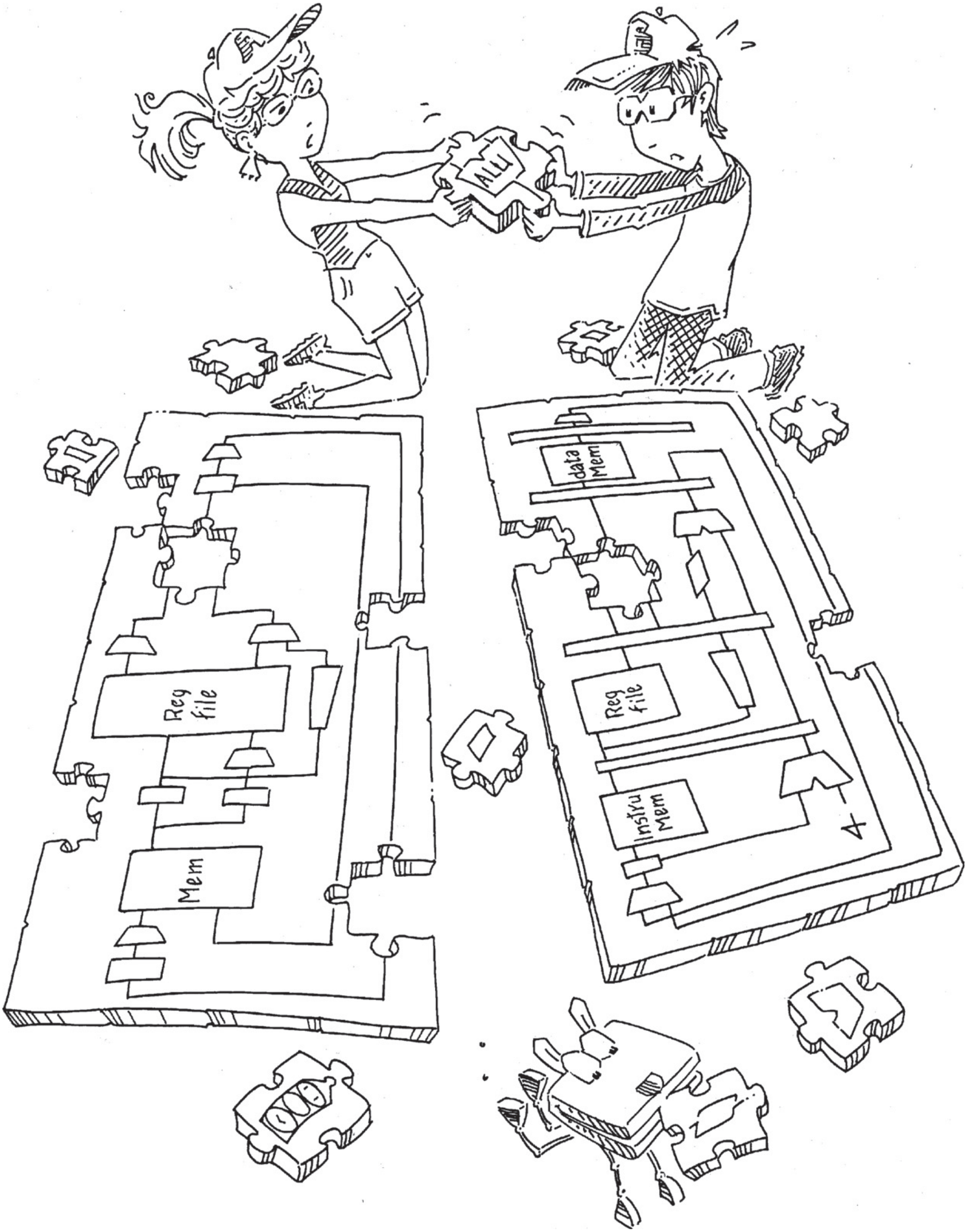
Question 6.3 You are given an array that holds a C string. The string forms a sentence. Design an algorithm for reversing the words in the sentence and storing the new sentence back in the array. Implement your algorithm using MIPS assembly code.

Question 6.4 Design an algorithm for counting the number of 1's in a 32-bit number. Implement your algorithm using MIPS assembly code.

Question 6.5 Write MIPS assembly code to reverse the bits in a register. Use as few instructions as possible. Assume the register of interest is `$t3`.

Question 6.6 Write MIPS assembly code to test whether overflow occurs when `$t2` and `$t3` are added. Use a minimum number of instructions.

Question 6.7 Design an algorithm for testing whether a given string is a palindrome. (Recall that a palindrome is a word that is the same forward and backward. For example, the words "wow" and "racecar" are palindromes.) Implement your algorithm using MIPS assembly code.

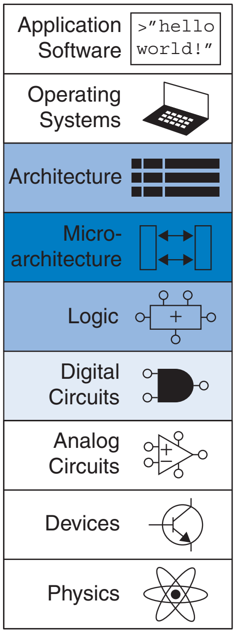

ALU
data Mem
Reg file
4
Instru Mem
Reg file
Mem

7

Microarchitecture

With contributions from Matthew Watkins

7.1 **Introduction**
7.2 **Performance Analysis**
7.3 **Single-Cycle Processor**
7.4 **Multicycle Processor**
7.5 **Pipelined Processor**
7.6 **HDL Representation***
7.7 **Exceptions***
7.8 **Advanced Microarchitecture***
7.9 **Real-World Perspective: x86 Microarchitecture***
7.10 **Summary**
Exercises
Interview Questions



7.1 INTRODUCTION

In this chapter, you will learn how to piece together a MIPS microprocessor. Indeed, you will puzzle out three different versions, each with different trade-offs between performance, cost, and complexity.

To the uninitiated, building a microprocessor may seem like black magic. But it is actually relatively straightforward, and by this point you have learned everything you need to know. Specifically, you have learned to design combinational and sequential logic given functional and timing specifications. You are familiar with circuits for arithmetic and memory. And you have learned about the MIPS architecture, which specifies the programmer's view of the MIPS processor in terms of registers, instructions, and memory.

This chapter covers *microarchitecture*, which is the connection between logic and architecture. Microarchitecture is the specific arrangement of registers, ALUs, finite state machines (FSMs), memories, and other logic building blocks needed to implement an architecture. A particular architecture, such as MIPS, may have many different microarchitectures, each with different trade-offs of performance, cost, and complexity. They all run the same programs, but their internal designs vary widely. We will design three different microarchitectures in this chapter to illustrate the trade-offs.

This chapter draws heavily on David Patterson and John Hennessy's classic MIPS designs in their text *Computer Organization and Design*. They have generously shared their elegant designs, which have the virtue of illustrating a real commercial architecture while being relatively simple and easy to understand.

7.1.1 Architectural State and Instruction Set

Recall that a computer architecture is defined by its instruction set and *architectural state*. The architectural state for the MIPS processor consists

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00007-0
© 2013 Elsevier, Inc. All rights reserved.

David Patterson was the first in his family to graduate from college (UCLA, 1969). He has been a professor of computer science at UC Berkeley since 1977, where he coinvented RISC, the Reduced Instruction Set Computer. In 1984, he developed the SPARC architecture used by Sun Microsystems. He is also the father of *RAID* (*Redundant Array of Inexpensive Disks*) and NOW (*Network of Workstations*).

John Hennessy is president of Stanford University and has been a professor of electrical engineering and computer science there since 1977. He coinvented RISC. He developed the MIPS architecture at Stanford in 1984 and cofounded MIPS Computer Systems. As of 2004, more than 300 million MIPS microprocessors have been sold.

In their copious free time, these two modern paragons write textbooks for recreation and relaxation.

of the program counter and the 32 registers. Any MIPS microarchitecture must contain all of this state. Based on the current architectural state, the processor executes a particular instruction with a particular set of data to produce a new architectural state. Some microarchitectures contain additional *nonarchitectural state* to either simplify the logic or improve performance; we will point this out as it arises.

To keep the microarchitectures easy to understand, we consider only a subset of the MIPS instruction set. Specifically, we handle the following instructions:

- R-type arithmetic/logic instructions: `add`, `sub`, `and`, `or`, `slt`
- Memory instructions: `lw`, `sw`
- Branches: `beq`

After building the microarchitectures with these instructions, we extend them to handle `addi` and `j`. These particular instructions were chosen because they are sufficient to write many interesting programs. Once you understand how to implement these instructions, you can expand the hardware to handle others.

7.1.2 Design Process

We will divide our microarchitectures into two interacting parts: the *datapath* and the *control*. The datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers. MIPS is a 32-bit architecture, so we will use a 32-bit datapath. The control unit receives the current instruction from the datapath and tells the datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the datapath.

A good way to design a complex system is to start with hardware containing the state elements. These elements include the memories and the architectural state (the program counter and registers). Then, add blocks of combinational logic between the state elements to compute the new state based on the current state. The instruction is read from part of memory; load and store instructions then read or write data from another part of memory. Hence, it is often convenient to partition the overall memory into two smaller memories, one containing instructions and the other containing data. Figure 7.1 shows a block diagram with the four state elements: the program counter, register file, and instruction and data memories.

In Figure 7.1, heavy lines are used to indicate 32-bit data busses. Medium lines are used to indicate narrower busses, such as the 5-bit address busses on the register file. Narrow blue lines are used to indicate

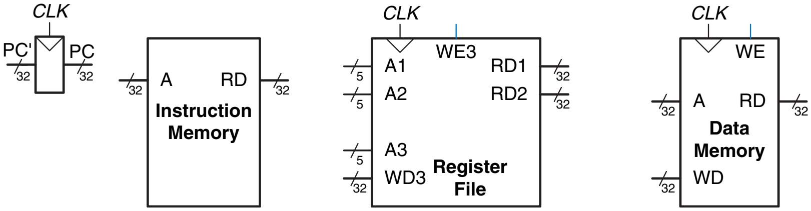


Figure 7.1 State elements of MIPS processor

control signals, such as the register file write enable. We will use this convention throughout the chapter to avoid cluttering diagrams with bus widths. Also, state elements usually have a reset input to put them into a known state at start-up. Again, to save clutter, this reset is not shown.

The *program counter* is an ordinary 32-bit register. Its output, *PC*, points to the current instruction. Its input, *PC′*, indicates the address of the next instruction.

The *instruction memory* has a single read port.[1] It takes a 32-bit instruction address input, *A*, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, *RD*.

The 32-element × 32-bit *register file* has two read ports and one write port. The read ports take 5-bit address inputs, *A1* and *A2*, each specifying one of $2^5 = 32$ registers as source operands. They read the 32-bit register values onto read data outputs *RD1* and *RD2*, respectively. The write port takes a 5-bit address input, *A3*; a 32-bit write data input, *WD*; a write enable input, *WE3*; and a clock. If the write enable is 1, the register file writes the data into the specified register on the rising edge of the clock.

The *data memory* has a single read/write port. If the write enable, *WE*, is 1, it writes data *WD* into address *A* on the rising edge of the clock. If the write enable is 0, it reads address *A* onto *RD*.

The instruction memory, register file, and data memory are all read *combinationally*. In other words, if the address changes, the new data appears at *RD* after some propagation delay; no clock is involved. They are written only on the rising edge of the clock. In this fashion, the state of the system is changed only at the clock edge. The address, data, and write enable must setup sometime before the clock edge and must remain stable until a hold time after the clock edge.

Because the state elements change their state only on the rising edge of the clock, they are synchronous sequential circuits. The microprocessor

Resetting the PC

At the very least, the program counter must have a reset signal to initialize its value when the processor turns on. MIPS processors initialize the PC to `0xBFC00000` on reset and begin executing code to start up the operating system (OS). The OS then loads an application program at `0x00400000` and begins executing it. For simplicity in this chapter, we will reset the PC to `0x00000000` and place our programs there instead.

---

[1] This is an oversimplification used to treat the instruction memory as a ROM; in most real processors, the instruction memory must be writable so that the OS can load a new program into memory. The multicycle microarchitecture described in Section 7.4 is more realistic in that it uses a combined memory for instructions and data that can be both read and written.

is built of clocked state elements and combinational logic, so it too is a synchronous sequential circuit. Indeed, the processor can be viewed as a giant finite state machine, or as a collection of simpler interacting state machines.

7.1.3 MIPS Microarchitectures

In this chapter, we develop three microarchitectures for the MIPS processor architecture: single-cycle, multicycle, and pipelined. They differ in the way that the state elements are connected together and in the amount of nonarchitectural state.

The *single-cycle microarchitecture* executes an entire instruction in one cycle. It is easy to explain and has a simple control unit. Because it completes the operation in one cycle, it does not require any nonarchitectural state. However, the cycle time is limited by the slowest instruction.

The *multicycle microarchitecture* executes instructions in a series of shorter cycles. Simpler instructions execute in fewer cycles than complicated ones. Moreover, the multicycle microarchitecture reduces the hardware cost by reusing expensive hardware blocks such as adders and memories. For example, the adder may be used on several different cycles for several purposes while carrying out a single instruction. The multicycle microprocessor accomplishes this by adding several nonarchitectural registers to hold intermediate results. The multicycle processor executes only one instruction at a time, but each instruction takes multiple clock cycles.

The *pipelined microarchitecture* applies pipelining to the single-cycle microarchitecture. It therefore can execute several instructions simultaneously, improving the throughput significantly. Pipelining must add logic to handle dependencies between simultaneously executing instructions. It also requires nonarchitectural pipeline registers. The added logic and registers are worthwhile; all commercial high-performance processors use pipelining today.

We explore the details and trade-offs of these three microarchitectures in the subsequent sections. At the end of the chapter, we briefly mention additional techniques that are used to get even more speed in modern high-performance microprocessors.

7.2 PERFORMANCE ANALYSIS

As we mentioned, a particular processor architecture can have many microarchitectures with different cost and performance trade-offs. The cost depends on the amount of hardware required and the implementation technology. Each year, CMOS processes can pack more transistors on a chip for the same amount of money, and processors take advantage of these additional transistors to deliver more performance. Precise cost calculations require detailed knowledge of the implementation technology, but in general, more gates and more memory mean more dollars. This section lays the foundation for analyzing performance.

There are many ways to measure the performance of a computer system, and marketing departments are infamous for choosing the method that makes their computer look fastest, regardless of whether the measurement has any correlation to real-world performance. For example, Intel and Advanced Micro Devices (AMD) both sell compatible microprocessors conforming to the x86 architecture. Intel Pentium III and Pentium 4 microprocessors were largely advertised according to clock frequency in the late 1990s and early 2000s, because Intel offered higher clock frequencies than its competitors. However, Intel's main competitor, AMD, sold Athlon microprocessors that executed programs faster than Intel's chips at the same clock frequency. What is a consumer to do?

The only gimmick-free way to measure performance is by measuring the execution time of a program of interest to you. The computer that executes your program fastest has the highest performance. The next best choice is to measure the total execution time of a collection of programs that are similar to those you plan to run; this may be necessary if you haven't written your program yet or if somebody else who doesn't have your program is making the measurements. Such collections of programs are called *benchmarks*, and the execution times of these programs are commonly published to give some indication of how a processor performs.

The execution time of a program, measured in seconds, is given by Equation 7.1.

$$\mathit{Execution\ Time} = \left(\#\,\mathit{instructions}\right)\left(\frac{\mathit{cycles}}{\mathit{instruction}}\right)\left(\frac{\mathit{seconds}}{\mathit{cycle}}\right) \tag{7.1}$$

The number of instructions in a program depends on the processor architecture. Some architectures have complicated instructions that do more work per instruction, thus reducing the number of instructions in a program. However, these complicated instructions are often slower to execute in hardware. The number of instructions also depends enormously on the cleverness of the programmer. For the purposes of this chapter, we will assume that we are executing known programs on a MIPS processor, so the number of instructions for each program is constant, independent of the microarchitecture.

The number of cycles per instruction, often called *CPI*, is the number of clock cycles required to execute an average instruction. It is the reciprocal of the throughput (instructions per cycle, or *IPC*). Different microarchitectures have different CPIs. In this chapter, we will assume we have an ideal memory system that does not affect the CPI. In Chapter 8, we examine how the processor sometimes has to wait for the memory, which increases the CPI.

The number of seconds per cycle is the clock period, T_c. The clock period is determined by the critical path through the logic on the processor. Different microarchitectures have different clock periods. Logic and circuit designs also significantly affect the clock period. For example, a carry-lookahead adder is faster than a ripple-carry adder. Manufacturing advances have historically doubled transistor speeds every 4–6 years, so a microprocessor built today will be much faster than one from last decade, even if the microarchitecture and logic are unchanged.

The challenge of the microarchitect is to choose the design that minimizes the execution time while satisfying constraints on cost and/or power consumption. Because microarchitectural decisions affect both CPI and T_c and are influenced by logic and circuit designs, determining the best choice requires careful analysis.

There are many other factors that affect overall computer performance. For example, the hard disk, the memory, the graphics system, and the network connection may be limiting factors that make processor performance irrelevant. The fastest microprocessor in the world doesn't help surfing the Internet on a dial-up connection. But these other factors are beyond the scope of this book.

7.3 SINGLE-CYCLE PROCESSOR

We first design a MIPS microarchitecture that executes instructions in a single cycle. We begin constructing the datapath by connecting the state elements from Figure 7.1 with combinational logic that can execute the various instructions. Control signals determine which specific instruction is carried out by the datapath at any given time. The controller contains combinational logic that generates the appropriate control signals based on the current instruction. We conclude by analyzing the performance of the single-cycle processor.

7.3.1 Single-Cycle Datapath

This section gradually develops the single-cycle datapath, adding one piece at a time to the state elements from Figure 7.1. The new connections are emphasized in black (or blue, for new control signals), while the hardware that has already been studied is shown in gray.

The program counter (PC) register contains the address of the instruction to execute. The first step is to read this instruction from instruction memory. Figure 7.2 shows that the PC is simply connected to the address input of the instruction memory. The instruction memory reads out, or *fetches*, the 32-bit instruction, labeled *Instr*.

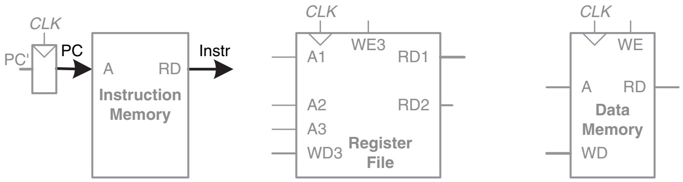

CLK
PC'
PC
A
RD
Instruction Memory
Instr
CLK
A1
WE3
RD1
A2
RD2
A3
WD3
Register File
CLK
WE
A
RD
Data Memory
WD

Figure 7.2 Fetch instruction from memory

The processor's actions depend on the specific instruction that was fetched. First we will work out the datapath connections for the `lw` instruction. Then we will consider how to generalize the datapath to handle the other instructions.

For a `lw` instruction, the next step is to read the source register containing the base address. This register is specified in the `rs` field of the instruction, $Instr_{25:21}$. These bits of the instruction are connected to the address input of one of the register file read ports, $A1$, as shown in Figure 7.3. The register file reads the register value onto $RD1$.

The `lw` instruction also requires an offset. The offset is stored in the immediate field of the instruction, $Instr_{15:0}$. Because the 16-bit immediate might be either positive or negative, it must be sign-extended to 32 bits, as shown in Figure 7.4. The 32-bit sign-extended value is called $SignImm$. Recall from Section 1.4.6 that sign extension simply copies the sign bit (most significant bit) of a short input into all of the upper bits of the longer output. Specifically, $SignImm_{15:0} = Instr_{15:0}$ and $SignImm_{31:16} = Instr_{15}$.

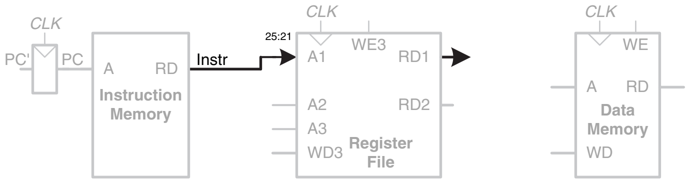

CLK
PC'
PC
A
RD
Instruction Memory
Instr
25:21
CLK
A1
WE3
RD1
A2
RD2
A3
WD3
Register File
CLK
WE
A
RD
Data Memory
WD

Figure 7.3 Read source operand from register file

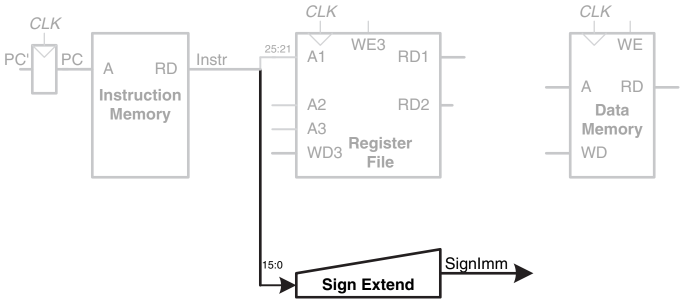

CLK
PC'
PC
A
RD
Instruction Memory
Instr
25:21
CLK
A1
WE3
RD1
A2
RD2
A3
WD3
Register File
CLK
WE
A
RD
Data Memory
WD
15:0
Sign Extend
SignImm

Figure 7.4 Sign-extend the immediate

The processor must add the base address to the offset to find the address to read from memory. Figure 7.5 introduces an ALU to perform this addition. The ALU receives two operands, *SrcA* and *SrcB*. *SrcA* comes from the register file, and *SrcB* comes from the sign-extended immediate. The ALU can perform many operations, as was described in Section 5.2.4. The 3-bit *ALUControl* signal specifies the operation. The ALU generates a 32-bit *ALUResult* and a *Zero* flag, that indicates whether *ALUResult* == 0. For a `lw` instruction, the *ALUControl* signal should be set to 010 to add the base address and offset. *ALUResult* is sent to the data memory as the address for the load instruction, as shown in Figure 7.5.

The data is read from the data memory onto the *ReadData* bus, then written back to the destination register in the register file at the end of the cycle, as shown in Figure 7.6. Port 3 of the register file is the write port.

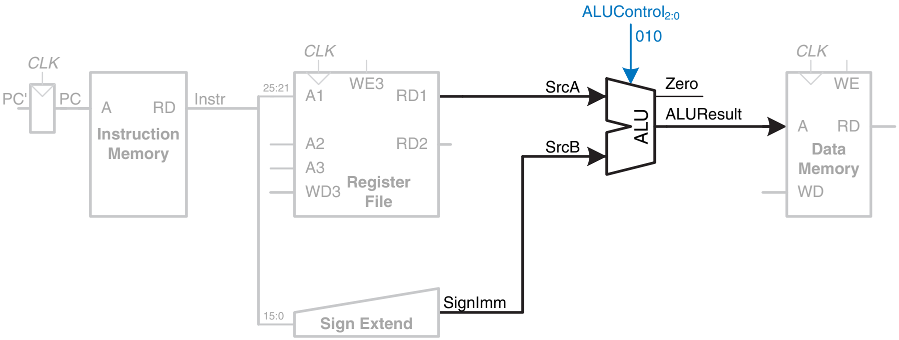

ALUControl$_{2:0}$
010
CLK
PC'
PC
A
RD
Instruction Memory
Instr
25:21
CLK
A1
WE3
RD1
A2
RD2
A3
WD3
Register File
SrcA
SrcB
ALU
Zero
ALUResult
CLK
WE
A
RD
Data Memory
WD
15:0
Sign Extend
SignImm

Figure 7.5 Compute memory address

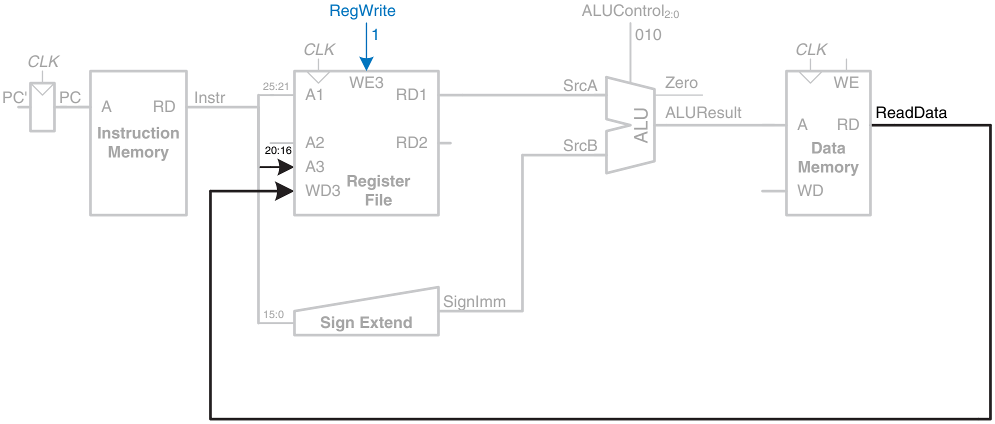

RegWrite
1
ALUControl$_{2:0}$
010
CLK
PC'
PC
A
RD
Instruction Memory
Instr
25:21
20:16
CLK
A1
WE3
RD1
A2
RD2
A3
WD3
Register File
SrcA
SrcB
ALU
Zero
ALUResult
CLK
WE
A
RD
Data Memory
WD
ReadData
15:0
Sign Extend
SignImm

Figure 7.6 Write data back to register file

The destination register for the `lw` instruction is specified in the `rt` field, $Instr_{20:16}$, which is connected to the port 3 address input, *A3*, of the register file. The *ReadData* bus is connected to the port 3 write data input, *WD3*, of the register file. A control signal called *RegWrite* is connected to the port 3 write enable input, *WE3*, and is asserted during a `lw` instruction so that the data value is written into the register file. The write takes place on the rising edge of the clock at the end of the cycle.

While the instruction is being executed, the processor must compute the address of the next instruction, *PC′*. Because instructions are 32 bits = 4 bytes, the next instruction is at $PC + 4$. Figure 7.7 uses another adder to increment the PC by 4. The new address is written into the program counter on the next rising edge of the clock. This completes the datapath for the `lw` instruction.

Next, let us extend the datapath to also handle the `sw` instruction. Like the `lw` instruction, the `sw` instruction reads a base address from port 1 of the register file and sign-extends an immediate. The ALU adds the base address to the immediate to find the memory address. All of these functions are already supported by the datapath.

The `sw` instruction also reads a second register from the register file and writes it to the data memory. Figure 7.8 shows the new connections for this function. The register is specified in the `rt` field, $Instr_{20:16}$. These bits of the instruction are connected to the second register file read port, *A2*. The register value is read onto the *RD2* port. It is connected to the write data port of the data memory. The write enable port of the data memory, *WE*, is controlled by *MemWrite*. For a `sw` instruction, $MemWrite = 1$, to write the data to memory; $ALUControl = 010$, to add the base address and offset; and $RegWrite = 0$, because nothing should

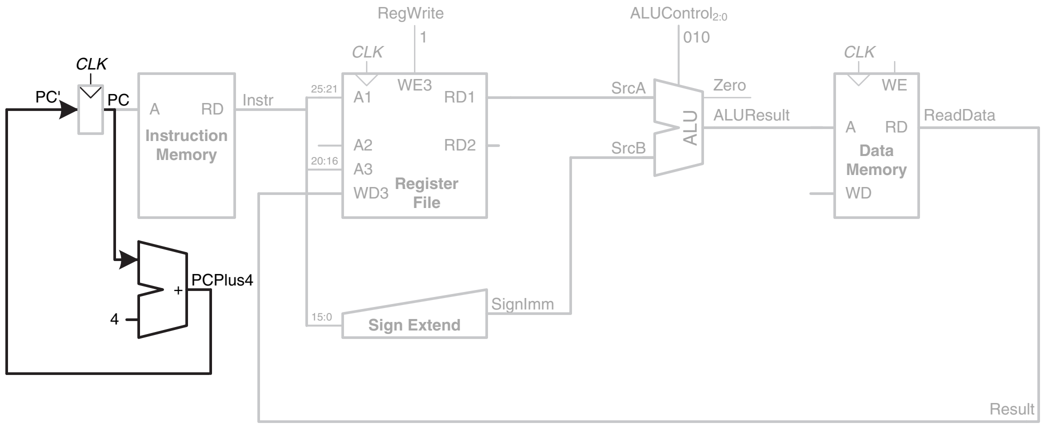

RegWrite
1
ALUControl$_{2:0}$
010
CLK
CLK
CLK
PC'
PC
A
RD
Instruction Memory
Instr
25:21
A1
WE3
RD1
A2
RD2
20:16
A3
WD3
Register File
SrcA
SrcB
ALU
Zero
ALUResult
WE
A
RD
Data Memory
WD
ReadData
+
PCPlus4
4
15:0
Sign Extend
SignImm
Result

Figure 7.7 Determine address of next instruction for PC

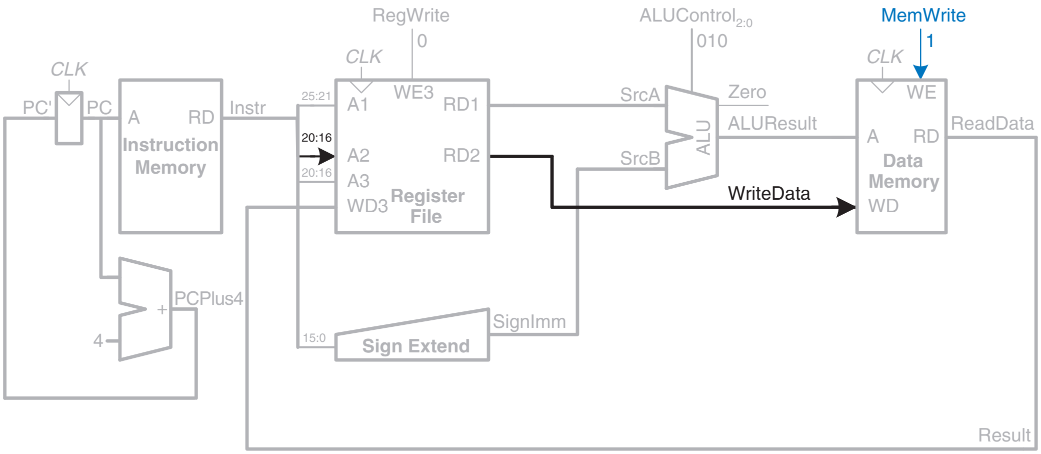

RegWrite
0
ALUControl$_{2:0}$
010
MemWrite
1
CLK
CLK
CLK
PC'
PC
A
RD
Instruction Memory
Instr
25:21
A1
WE3
RD1
20:16
A2
RD2
20:16
A3
WD3
Register File
SrcA
SrcB
ALU
Zero
ALUResult
WriteData
WE
A
RD
Data Memory
WD
ReadData
PCPlus4
+
4
15:0
Sign Extend
SignImm
Result

Figure 7.8 **Write data to memory for `sw` instruction**

be written to the register file. Note that data is still read from the address given to the data memory, but that this *ReadData* is ignored because *RegWrite* = 0.

Next, consider extending the datapath to handle the R-type instructions `add`, `sub`, `and`, `or`, and `slt`. All of these instructions read two registers from the register file, perform some ALU operation on them, and write the result back to a third register file. They differ only in the specific ALU operation. Hence, they can all be handled with the same hardware, using different *ALUControl* signals.

Figure 7.9 shows the enhanced datapath handling R-type instructions. The register file reads two registers. The ALU performs an operation on these two registers. In Figure 7.8, the ALU always received its *SrcB* operand from the sign-extended immediate (*SignImm*). Now, we add a multiplexer to choose *SrcB* from either the register file *RD2* port or *SignImm*.

The multiplexer is controlled by a new signal, *ALUSrc*. *ALUSrc* is 0 for R-type instructions to choose *SrcB* from the register file; it is 1 for `lw` and `sw` to choose *SignImm*. This principle of enhancing the datapath's capabilities by adding a multiplexer to choose inputs from several possibilities is extremely useful. Indeed, we will apply it twice more to complete the handling of R-type instructions.

In Figure 7.8, the register file always got its write data from the data memory. However, R-type instructions write the *ALUResult* to the register file. Therefore, we add another multiplexer to choose between *ReadData* and *ALUResult*. We call its output *Result*. This multiplexer is controlled by another new signal, *MemtoReg*. *MemtoReg* is 0 for R-type

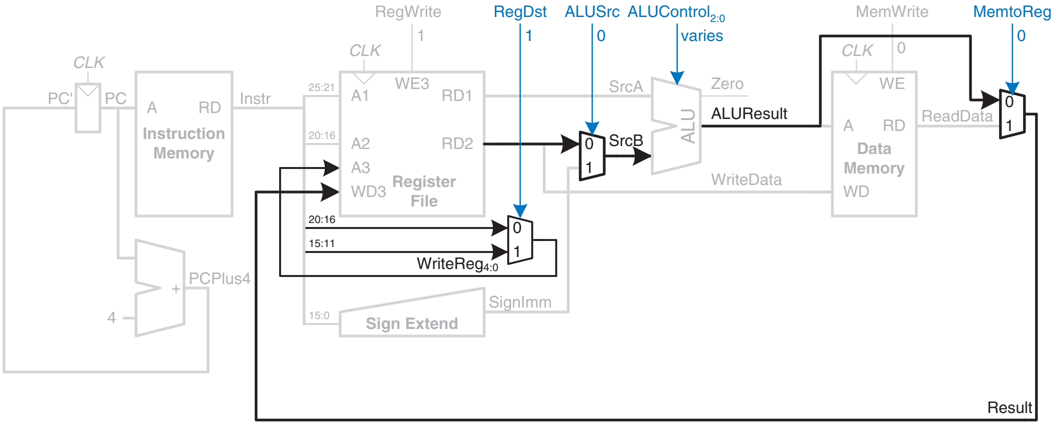

RegWrite
1
RegDst
1
ALUSrc
0
ALUControl$_{2:0}$
varies
MemWrite
0
MemtoReg
0
CLK
PC'
PC
A
RD
Instruction Memory
Instr
25:21
20:16
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
20:16
15:11
WriteReg$_{4:0}$
0
1
SrcA
SrcB
ALU
Zero
ALUResult
WriteData
WE
A
RD
Data Memory
WD
ReadData
PCPlus4
4
+
15:0
Sign Extend
SignImm
Result

Figure 7.9 **Datapath enhancements for R-type instruction**

instructions to choose *Result* from the *ALUResult*; it is 1 for `lw` to choose *ReadData*. We don't care about the value of *MemtoReg* for `sw`, because `sw` does not write to the register file.

Similarly, in Figure 7.8, the register to write was specified by the `rt` field of the instruction, $Instr_{20:16}$. However, for R-type instructions, the register is specified by the `rd` field, $Instr_{15:11}$. Thus, we add a third multiplexer to choose *WriteReg* from the appropriate field of the instruction. The multiplexer is controlled by *RegDst*. *RegDst* is 1 for R-type instructions to choose *WriteReg* from the `rd` field, $Instr_{15:11}$; it is 0 for `lw` to choose the `rt` field, $Instr_{20:16}$. We don't care about the value of *RegDst* for `sw`, because `sw` does not write to the register file.

Finally, let us extend the datapath to handle `beq`. `beq` compares two registers. If they are equal, it takes the branch by adding the branch offset to the program counter. Recall that the offset is a positive or negative number, stored in the `imm` field of the instruction, $Instr_{15:0}$. The offset indicates the number of instructions to branch past. Hence, the immediate must be sign-extended and multiplied by 4 to get the new program counter value: $PC' = PC + 4 + SignImm \times 4$.

Figure 7.10 shows the datapath modifications. The next *PC* value for a taken branch, *PCBranch*, is computed by shifting *SignImm* left by 2 bits, then adding it to *PCPlus4*. The left shift by 2 is an easy way to multiply by 4, because a shift by a constant amount involves just wires. The two registers are compared by computing $SrcA - SrcB$ using the ALU. If *ALUResult* is 0, as indicated by the *Zero* flag from the ALU, the registers are equal. We add a multiplexer to choose *PC'* from either *PCPlus4* or *PCBranch*. *PCBranch* is selected if the instruction is a branch

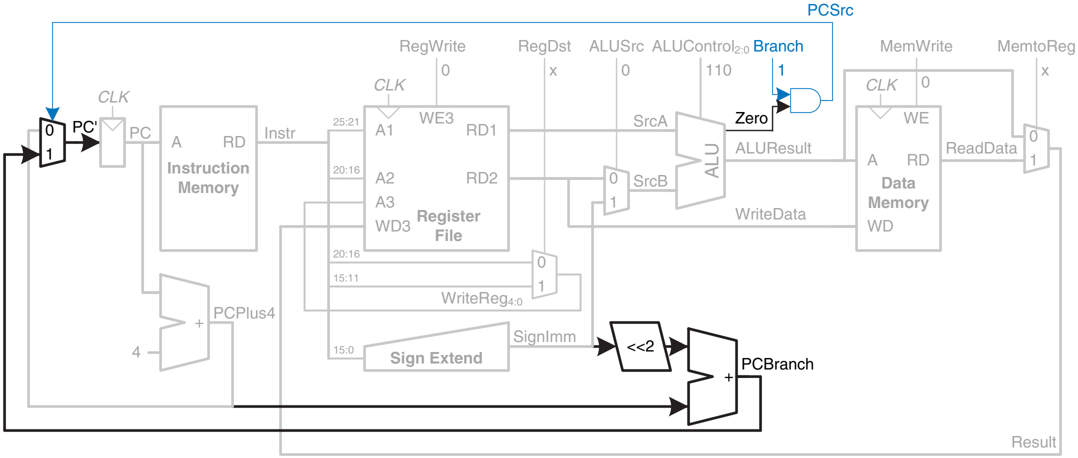

PCSrc
RegWrite
RegDst
ALUSrc
ALUControl2:0
Branch
MemWrite
MemtoReg
0
x
0
110
1
0
x
CLK
CLK
CLK
PC'
PC
0
1
A
RD
Instruction Memory
Instr
25:21
20:16
20:16
15:11
15:0
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
0
1
ALU
Zero
ALUResult
WriteData
WE
A
RD
Data Memory
WD
ReadData
0
1
0
1
WriteReg4:0
PCPlus4
4
+
Sign Extend
SignImm
<<2
+
PCBranch
Result

Figure 7.10 **Datapath enhancements for** `beq` **instruction**

and the *Zero* flag is asserted. Hence, *Branch* is 1 for `beq` and 0 for other instructions. For `beq`, *ALUControl* = 110, so the ALU performs a subtraction. *ALUSrc* = 0 to choose *SrcB* from the register file. *RegWrite* and *MemWrite* are 0, because a branch does not write to the register file or memory. We don't care about the values of *RegDst* and *MemtoReg*, because the register file is not written.

This completes the design of the single-cycle MIPS processor datapath. We have illustrated not only the design itself, but also the design process in which the state elements are identified and the combinational logic connecting the state elements is systematically added. In the next section, we consider how to compute the control signals that direct the operation of our datapath.

7.3.2 Single-Cycle Control

The control unit computes the control signals based on the `opcode` and `funct` fields of the instruction, $Instr_{31:26}$ and $Instr_{5:0}$. Figure 7.11 shows the entire single-cycle MIPS processor with the control unit attached to the datapath.

Most of the control information comes from the `opcode`, but R-type instructions also use the `funct` field to determine the ALU operation. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic, as shown in Figure 7.12. The main decoder computes most of the outputs from the `opcode`. It also determines a 2-bit *ALUOp* signal. The ALU decoder uses this *ALUOp* signal in conjunction with the `funct` field to compute *ALUControl*. The meaning of the *ALUOp* signal is given in Table 7.1.

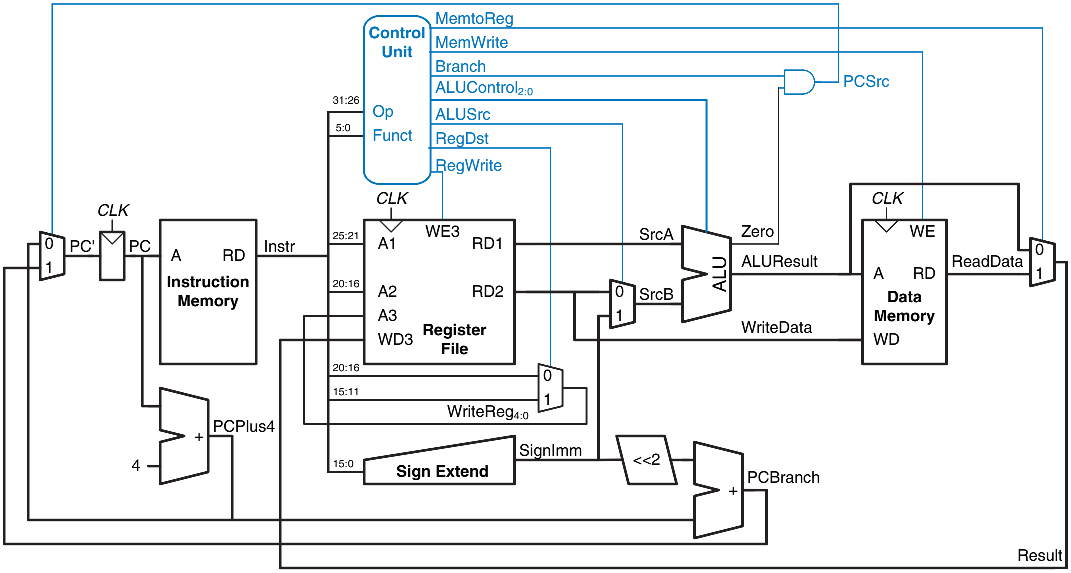

Control Unit
MemtoReg
MemWrite
Branch
ALUControl2:0
PCSrc
ALUSrc
RegDst
RegWrite
31:26
Op
5:0
Funct
CLK
CLK
CLK
WE3
WE
PC'
PC
A
RD
Instr
Instruction Memory
25:21
A1
RD1
SrcA
Zero
ALU
ALUResult
A
RD
ReadData
Data Memory
20:16
A2
RD2
SrcB
A3
WD3
Register File
WriteData
WD
20:16
15:11
WriteReg4:0
PCPlus4
4
+
15:0
Sign Extend
SignImm
<<2
PCBranch
+
Result
0
1

Figure 7.11 Complete single-cycle MIPS processor

Table 7.1 *ALUOp* **encoding**

ALUOp	Meaning
00	add
01	subtract
10	look at `funct` field
11	n/a

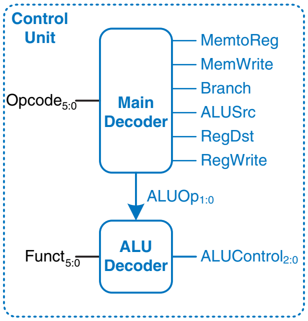

Control Unit
Opcode5:0
Main Decoder
MemtoReg
MemWrite
Branch
ALUSrc
RegDst
RegWrite
ALUOp1:0
Funct5:0
ALU Decoder
ALUControl2:0

Figure 7.12 Control unit internal structure

Table 7.2 is a truth table for the ALU decoder. Recall that the meanings of the three *ALUControl* signals were given in Table 5.1. Because *ALUOp* is never 11, the truth table can use don't care's X1 and 1X instead of 01 and 10 to simplify the logic. When *ALUOp* is 00 or 01, the ALU should add or subtract, respectively. When *ALUOp* is 10, the decoder examines the `funct` field to determine the *ALUControl*. Note that, for the R-type instructions we implement, the first two bits of the `funct` field are always 10, so we may ignore them to simplify the decoder.

The control signals for each instruction were described as we built the datapath. Table 7.3 is a truth table for the main decoder that summarizes the control signals as a function of the `opcode`. All R-type instructions use the same main decoder values; they differ only in the

Table 7.2 ALU decoder truth table

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (`add`)	010 (add)
1X	100010 (`sub`)	110 (subtract)
1X	100100 (`and`)	000 (and)
1X	100101 (`or`)	001 (or)
1X	101010 (`slt`)	111 (set less than)

Table 7.3 Main decoder truth table

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01

ALU decoder output. Recall that, for instructions that do not write to the register file (e.g., `sw` and `beq`), the *RegDst* and *MemtoReg* control signals are don't cares (X); the address and data to the register write port do not matter because *RegWrite* is not asserted. The logic for the decoder can be designed using your favorite techniques for combinational logic design.

Example 7.1 SINGLE-CYCLE PROCESSOR OPERATION

Determine the values of the control signals and the portions of the datapath that are used when executing an `or` instruction.

Solution: Figure 7.13 illustrates the control signals and flow of data during execution of the `or` instruction. The PC points to the memory location holding the instruction, and the instruction memory fetches this instruction.

The main flow of data through the register file and ALU is represented with a dashed blue line. The register file reads the two source operands specified by $Instr_{25:21}$ and $Instr_{20:16}$. *SrcB* should come from the second port of the register file

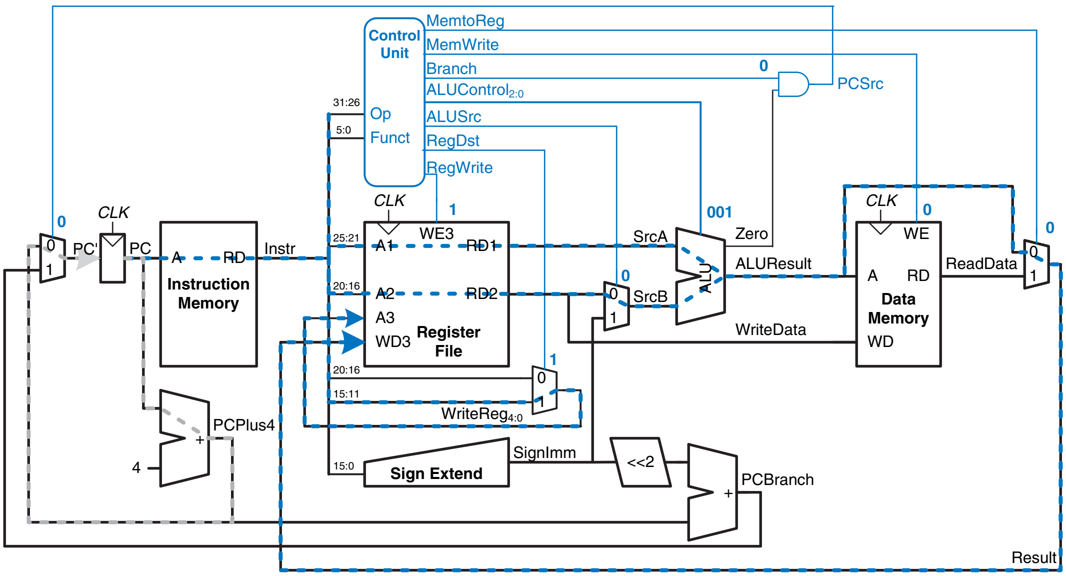

Control Unit
MemtoReg
MemWrite
Branch
ALUControl2:0
ALUSrc
RegDst
RegWrite
31:26
5:0
Op
Funct
0
PCSrc
CLK
1
001
CLK
0
WE
0
CLK
0
0
1
PC'
PC
A
RD
Instr
Instruction Memory
25:21
A1
WE3
RD1
SrcA
Zero
ALU
ALUResult
20:16
A2
RD2
0
SrcB
1
0
A3
WD3
Register File
WriteData
A
RD
Data Memory
WD
ReadData
0
1
20:16
15:11
1
0
1
WriteReg4:0
PCPlus4
4
+
15:0
Sign Extend
SignImm
<<2
+
PCBranch
Result

Figure 7.13 Control signals and data flow while executing `or` instruction

(not *SignImm*), so *ALUSrc* must be 0. `or` is an R-type instruction, so *ALUOp* is 10, indicating that *ALUControl* should be determined from the `funct` field to be 001. *Result* is taken from the ALU, so *MemtoReg* is 0. The result is written to the register file, so *RegWrite* is 1. The instruction does not write memory, so *MemWrite* = 0.

The selection of the destination register is also shown with a dashed blue line. The destination register is specified in the `rd` field, $Instr_{15:11}$, so *RegDst* = 1.

The updating of the PC is shown with the dashed gray line. The instruction is not a branch, so *Branch* = 0 and, hence, *PCSrc* is also 0. The PC gets its next value from *PCPlus4*.

Note that data certainly does flow through the nonhighlighted paths, but that the value of that data is unimportant for this instruction. For example, the immediate is sign-extended and data is read from memory, but these values do not influence the next state of the system.

7.3.3 More Instructions

We have considered a limited subset of the full MIPS instruction set. Adding support for the `addi` and `j` instructions illustrates the principle of how to handle new instructions and also gives us a sufficiently rich instruction set to write many interesting programs. We will see that supporting some instructions simply requires enhancing the main decoder, whereas supporting others also requires more hardware in the datapath.

Example 7.2 `addi` INSTRUCTION

The add immediate instruction, `addi`, adds the value in a register to the immediate and writes the result to another register. The datapath already is capable of this task. Determine the necessary changes to the controller to support `addi`.

Solution: All we need to do is add a new row to the main decoder truth table showing the control signal values for `addi`, as given in Table 7.4. The result should be written to the register file, so $RegWrite = 1$. The destination register is specified in the `rt` field of the instruction, so $RegDst = 0$. $SrcB$ comes from the immediate, so $ALUSrc = 1$. The instruction is not a branch, nor does it write memory, so $Branch = MemWrite = 0$. The result comes from the ALU, not memory, so $MemtoReg = 0$. Finally, the ALU should add, so $ALUOp = 00$.

Table 7.4 **Main decoder truth table enhanced to support** `addi`

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01
`addi`	001000	1	0	1	0	0	0	00

Example 7.3 `j` INSTRUCTION

The jump instruction, `j`, writes a new value into the PC. The two least significant bits of the PC are always 0, because the PC is word aligned (i.e., always a multiple of 4). The next 26 bits are taken from the jump address field in $Instr_{25:0}$. The upper four bits are taken from the old value of the PC.

The existing datapath lacks hardware to compute PC' in this fashion. Determine the necessary changes to both the datapath and controller to handle `j`.

Solution: First, we must add hardware to compute the next PC value, PC', in the case of a `j` instruction and a multiplexer to select this next PC, as shown in Figure 7.14. The new multiplexer uses the new $Jump$ control signal.

Now we must add a row to the main decoder truth table for the `j` instruction and a column for the *Jump* signal, as shown in Table 7.5. The *Jump* control signal is 1 for the `j` instruction and 0 for all others. `j` does not write the register file or memory, so $RegWrite = MemWrite = 0$. Hence, we don't care about the computation done in the datapath, and $RegDst = ALUSrc = Branch = MemtoReg = ALUOp = X$.

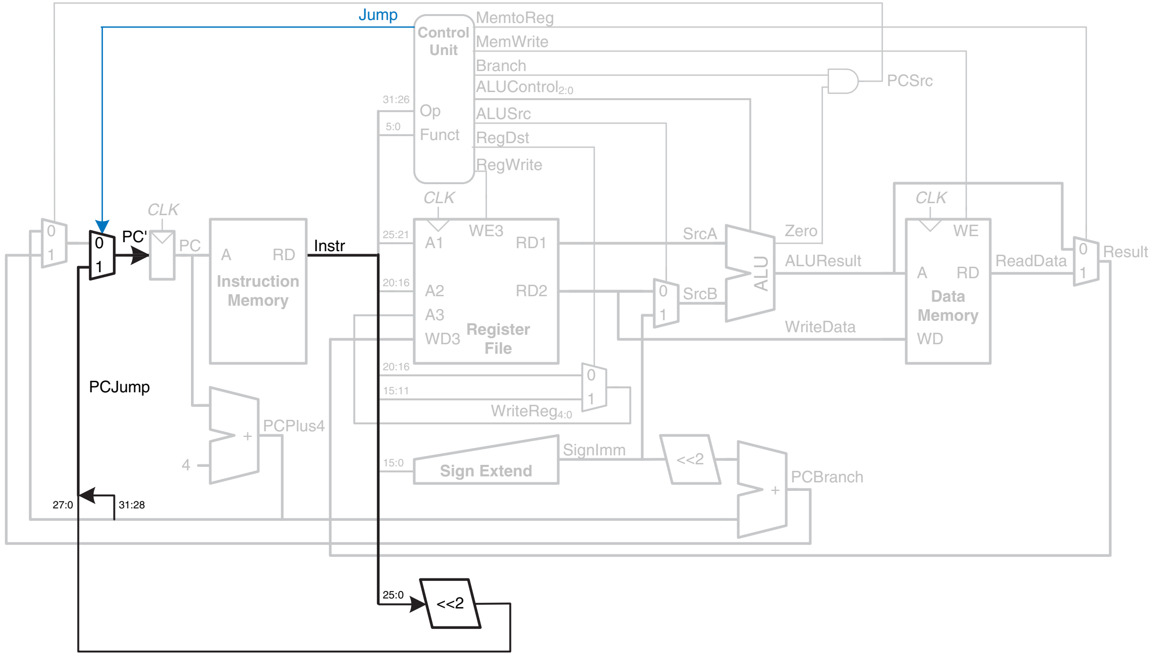

Jump
Control Unit
MemtoReg
MemWrite
Branch
ALUControl2:0
ALUSrc
RegDst
RegWrite
PCSrc
31:26
5:0
Op
Funct
CLK
CLK
CLK
0
1
0
1
PC'
PC
A
RD
Instruction Memory
Instr
25:21
20:16
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
20:16
15:11
0
1
WriteReg4:0
SrcA
SrcB
0
1
ALU
Zero
ALUResult
WriteData
WE
A
RD
Data Memory
WD
ReadData
0
1
Result
PCJump
+
4
PCPlus4
15:0
Sign Extend
SignImm
<<2
+
PCBranch
27:0
31:28
25:0
<<2

Figure 7.14 Single-cycle MIPS datapath enhanced to support the `j` instruction

Table 7.5 Main decoder truth table enhanced to support `j`

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
`lw`	100011	1	0	1	0	0	1	00	0
`sw`	101011	0	X	1	0	1	X	00	0
`beq`	000100	0	X	0	1	0	X	01	0
`addi`	001000	1	0	1	0	0	0	00	0
`j`	000010	0	X	X	X	0	X	XX	1

7.3.4 Performance Analysis

Each instruction in the single-cycle processor takes one clock cycle, so the CPI is 1. The critical path for the `lw` instruction is shown in Figure 7.15 with a heavy dashed blue line. It starts with the PC loading a new address on the rising edge of the clock. The instruction memory reads the next instruction. The register file reads *SrcA*. While the register file is reading, the immediate field is sign-extended and selected at the *ALUSrc* multiplexer to determine *SrcB*. The ALU adds *SrcA* and *SrcB* to find the effective address. The data memory reads from this address. The *MemtoReg* multiplexer selects *ReadData*. Finally, *Result* must setup at the register file before the next rising clock edge, so that it can be properly written. Hence, the cycle time is

$$\begin{aligned} T_c = t_{pcq\_\text{PC}} + t_{\text{mem}} + \max[t_{RF\text{read}}, t_{sext} + t_{\text{mux}}] \\ + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}} \end{aligned} \tag{7.2}$$

In most implementation technologies, the ALU, memory, and register file accesses are substantially slower than other operations. Therefore, the cycle time simplifies to

$$T_c = t_{pcq\_\text{PC}} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + t_{\text{mux}} + t_{RF\text{setup}} \tag{7.3}$$

The numerical values of these times will depend on the specific implementation technology.

Other instructions have shorter critical paths. For example, R-type instructions do not need to access data memory. However, we are

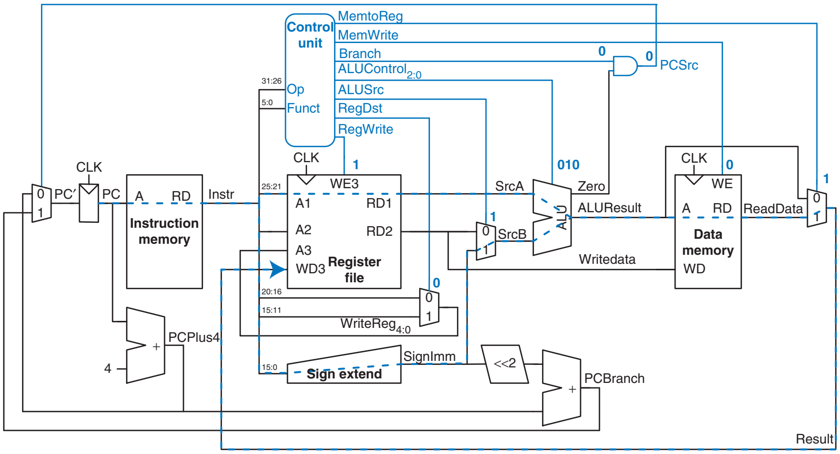

MemtoReg
Control unit
MemWrite
Branch
ALUControl2:0
ALUSrc
RegDst
RegWrite
Op
Funct
31:26
5:0
0
0
PCSrc
CLK
1
CLK
WE3
010
Zero
CLK
0
WE
1
PC'
PC
A
RD
Instr
Instruction memory
25:21
A1
RD1
SrcA
ALU
ALUResult
A
RD
ReadData
Data memory
WD
0
1
A2
RD2
1
0
1
SrcB
A3
WD3
Register file
Writedata
20:16
0
0
1
15:11
WriteReg4:0
PCPlus4
+
4
15:0
Sign extend
SignImm
<<2
+
PCBranch
Result

Figure 7.15 **Critical path for `lw` instruction**

disciplining ourselves to synchronous sequential design, so the clock period is constant and must be long enough to accommodate the slowest instruction.

Example 7.4 SINGLE-CYCLE PROCESSOR PERFORMANCE

Ben Bitdiddle is contemplating building the single-cycle MIPS processor in a 65 nm CMOS manufacturing process. He has determined that the logic elements have the delays given in Table 7.6. Help him compute the execution time for a program with 100 billion instructions.

Solution: According to Equation 7.3, the cycle time of the single-cycle processor is $T_{c1} = 30 + 2(250) + 150 + 200 + 25 + 20 = 925$ ps. We use the subscript "1" to distinguish it from subsequent processor designs. According to Equation 7.1, the total execution time is $T_1 = (100 \times 10^9$ instructions) (1 cycle/instruction) $(925 \times 10^{-12}$ s/cycle) $= 92.5$ seconds.

Table 7.6 **Delays of circuit elements**

Element	Parameter	Delay (ps)
register clk-to-Q	t_{pcq}	30
register setup	t_{setup}	20
multiplexer	t_{mux}	25
ALU	t_{ALU}	200
memory read	t_{mem}	250
register file read	$t_{RF\text{read}}$	150
register file setup	$t_{RF\text{setup}}$	20

7.4 MULTICYCLE PROCESSOR

The single-cycle processor has three primary weaknesses. First, it requires a clock cycle long enough to support the slowest instruction (`lw`), even though most instructions are faster. Second, it requires three adders (one in the ALU and two for the PC logic); adders are relatively expensive circuits, especially if they must be fast. And third, it has separate instruction and data memories, which may not be realistic. Most computers have a single large memory that holds both instructions and data and that can be read and written.

The multicycle processor addresses these weaknesses by breaking an instruction into multiple shorter steps. In each short step, the processor can read or write the memory or register file or use the ALU. Different instructions use different numbers of steps, so simpler instructions can complete faster than more complex ones. The processor needs only one adder; this adder is reused for different purposes on various steps. And the processor uses a combined memory for instructions and data. The instruction is fetched from memory on the first step, and data may be read or written on later steps.

We design a multicycle processor following the same procedure we used for the single-cycle processor. First, we construct a datapath by connecting the architectural state elements and memories with combinational logic. But, this time, we also add nonarchitectural state elements to hold intermediate results between the steps. Then we design the controller. The controller produces different signals on different steps during execution of a single instruction, so it is now a finite state machine rather than combinational logic. We again examine how to add new instructions to the processor. Finally, we analyze the performance of the multicycle processor and compare it to the single-cycle processor.

7.4.1 Multicycle Datapath

Again, we begin our design with the memory and architectural state of the MIPS processor, shown in Figure 7.16. In the single-cycle design, we used separate instruction and data memories because we needed to read the instruction memory and read or write the data memory all in one cycle. Now, we choose to use a combined memory for both instructions and data. This is more realistic, and it is feasible because we can read the instruction in one cycle, then read or write the data in a separate cycle. The PC and register file remain unchanged. We gradually build the datapath by adding components to handle each step of each instruction. The new connections are emphasized in black (or blue, for new control signals), whereas the hardware that has already been studied is shown in gray.

The PC contains the address of the instruction to execute. The first step is to read this instruction from instruction memory. Figure 7.17 shows that the PC is simply connected to the address input of the instruction memory. The instruction is read and stored in a new nonarchitectural

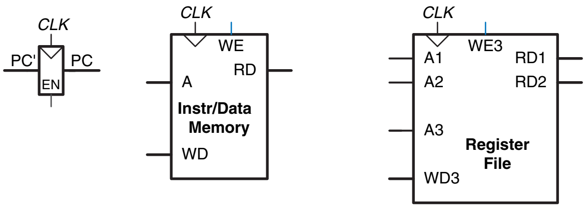


Figure 7.16 State elements with unified instruction/data memory

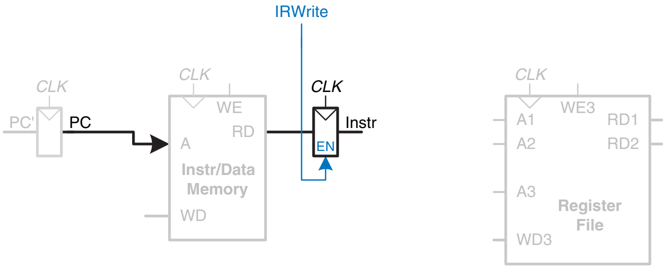

IRWrite
CLK
PC'
PC
CLK
WE
RD
A
Instr/Data
Memory
WD
CLK
Instr
EN
CLK
A1
A2
WE3
RD1
RD2
A3
Register
File
WD3

Figure 7.17 **Fetch instruction from memory**

Instruction Register so that it is available for future cycles. The Instruction Register receives an enable signal, called *IRWrite*, that is asserted when it should be updated with a new instruction.

As we did with the single-cycle processor, we will work out the datapath connections for the `lw` instruction. Then we will enhance the datapath to handle the other instructions. For a `lw` instruction, the next step is to read the source register containing the base address. This register is specified in the `rs` field of the instruction, $Instr_{25:21}$. These bits of the instruction are connected to one of the address inputs, *A1*, of the register file, as shown in Figure 7.18. The register file reads the register onto *RD1*. This value is stored in another nonarchitectural register, *A*.

The `lw` instruction also requires an offset. The offset is stored in the immediate field of the instruction, $Instr_{15:0}$, and must be sign-extended to 32 bits, as shown in Figure 7.19. The 32-bit sign-extended value is called *SignImm*. To be consistent, we might store *SignImm* in another nonarchitectural register. However, *SignImm* is a combinational function of *Instr* and will not change while the current instruction is being processed, so there is no need to dedicate a register to hold the constant value.

The address of the load is the sum of the base address and offset. We use an ALU to compute this sum, as shown in Figure 7.20. *ALUControl* should be set to 010 to perform an addition. *ALUResult* is stored in a nonarchitectural register called *ALUOut*.

The next step is to load the data from the calculated address in the memory. We add a multiplexer in front of the memory to choose the

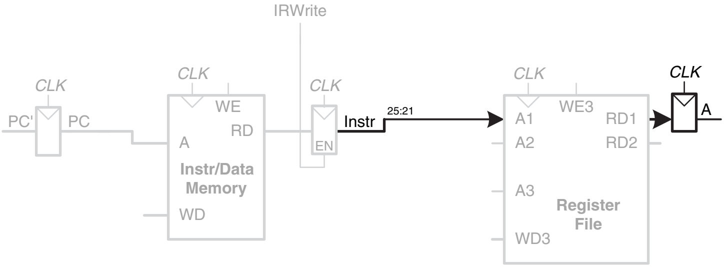

IRWrite
CLK
PC'
PC
CLK
WE
RD
A
Instr/Data
Memory
WD
CLK
EN
Instr
25:21
CLK
A1
A2
WE3
RD1
RD2
A3
Register
File
WD3
CLK
A

Figure 7.18 **Read source operand from register file**

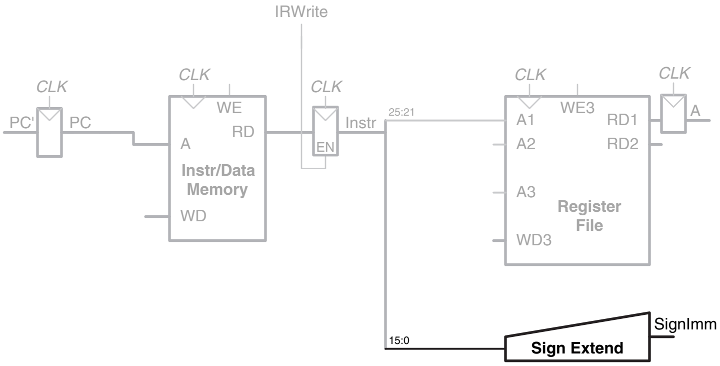

IRWrite
CLK
PC'
PC
CLK
WE
RD
A
Instr/Data
Memory
WD
CLK
EN
Instr
25:21
CLK
WE3
A1
A2
A3
WD3
RD1
RD2
Register
File
CLK
A
15:0
Sign Extend
SignImm

Figure 7.19 Sign-extend the immediate

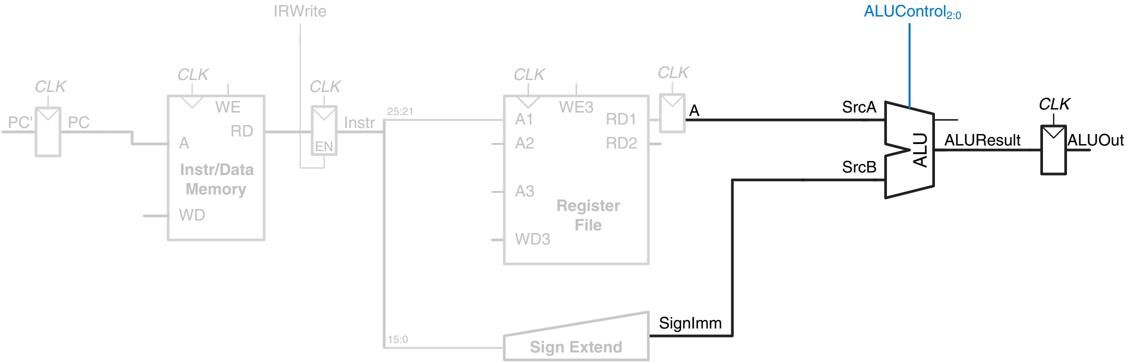

IRWrite
ALUControl$_{2:0}$
CLK
PC'
PC
CLK
WE
RD
A
Instr/Data
Memory
WD
CLK
EN
Instr
25:21
CLK
WE3
A1
A2
A3
WD3
RD1
RD2
Register
File
CLK
A
SrcA
SrcB
ALU
ALUResult
CLK
ALUOut
15:0
Sign Extend
SignImm

Figure 7.20 Add base address to offset

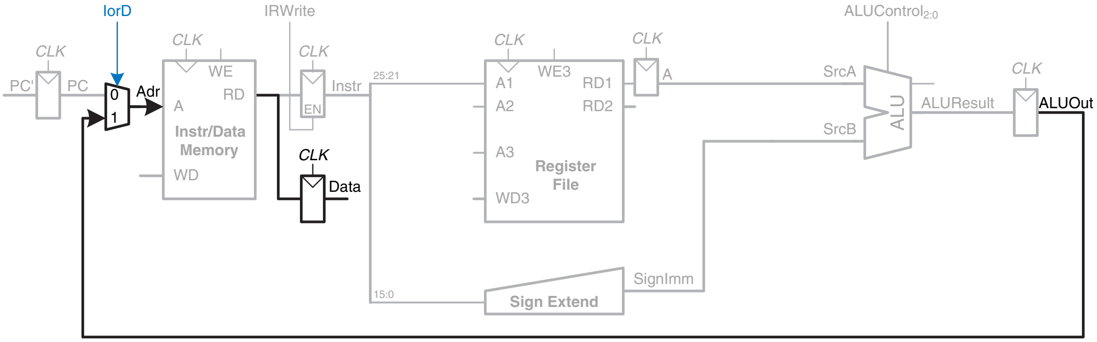

IorD
IRWrite
ALUControl$_{2:0}$
CLK
PC'
PC
0
1
Adr
CLK
WE
RD
A
Instr/Data
Memory
WD
CLK
EN
Instr
CLK
Data
25:21
CLK
WE3
A1
A2
A3
WD3
RD1
RD2
Register
File
CLK
A
SrcA
SrcB
ALU
ALUResult
CLK
ALUOut
15:0
Sign Extend
SignImm

Figure 7.21 Load data from memory

memory address, *Adr*, from either the PC or *ALUOut*, as shown in Figure 7.21. The multiplexer select signal is called *IorD*, to indicate either an instruction or data address. The data read from the memory is stored in another nonarchitectural register, called *Data*. Notice that the address multiplexer permits us to reuse the memory during the `lw` instruction. On the first step, the address is taken from the PC to fetch the instruction. On a later step, the address is taken from *ALUOut* to load the data. Hence, *IorD* must have different values on different steps. In Section 7.4.2, we develop the FSM controller that generates these sequences of control signals.

Finally, the data is written back to the register file, as shown in Figure 7.22. The destination register is specified by the `rt` field of the instruction, $Instr_{20:16}$.

While all this is happening, the processor must update the program counter by adding 4 to the old PC. In the single-cycle processor, a separate adder was needed. In the multicycle processor, we can use the existing ALU on one of the steps when it is not busy. To do so, we must insert source multiplexers to choose the PC and the constant 4 as ALU inputs, as shown in Figure 7.23. A two-input multiplexer controlled by *ALUSrcA* chooses either the PC or register *A* as *SrcA*. A four-input multiplexer controlled by *ALUSrcB* chooses either 4 or *SignImm* as *SrcB*. We use the other two multiplexer inputs later when we extend the datapath to handle other instructions. (The numbering of inputs to the multiplexer is arbitrary.) To update the PC, the ALU adds *SrcA* (*PC*) to *SrcB* (4), and the result is written into the program counter register. The *PCWrite* control signal enables the PC register to be written only on certain cycles.

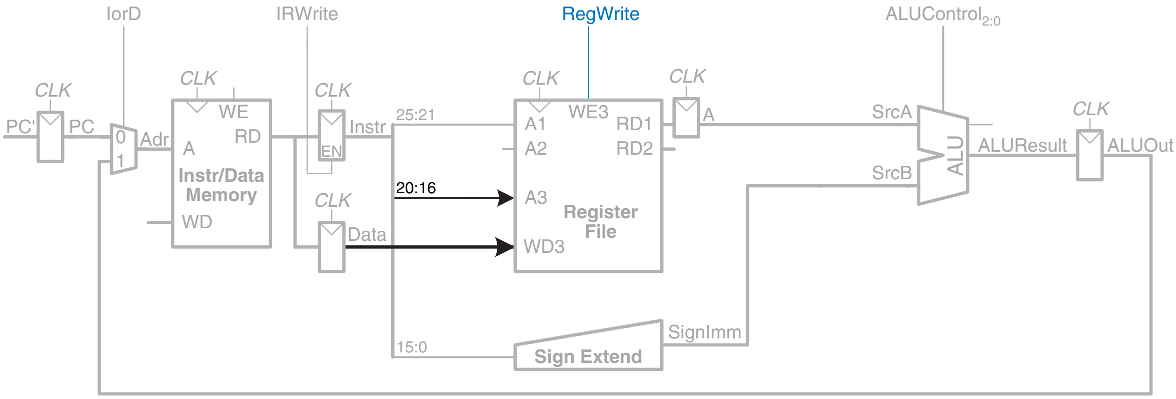


Figure 7.22 Write data back to register file

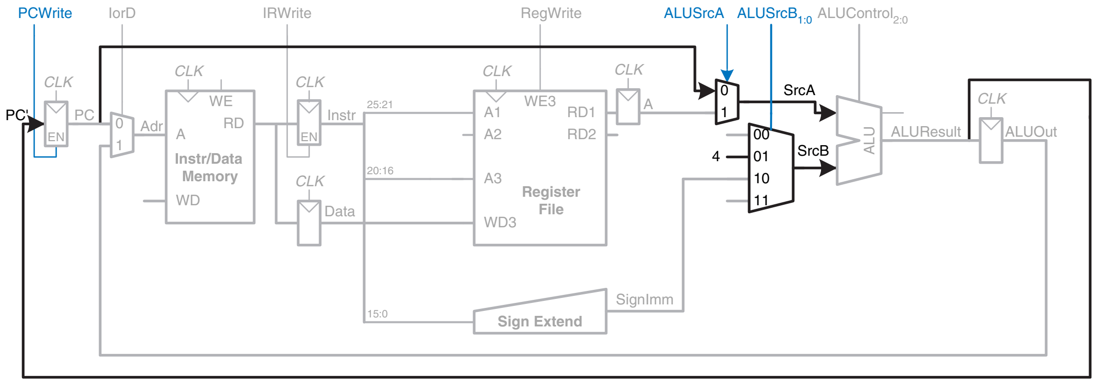

PCWrite
IorD
IRWrite
RegWrite
ALUSrcA
ALUSrcB1:0
ALUControl2:0
CLK
PC'
EN
PC
0
1
Adr
WE
A
RD
Instr/Data
Memory
WD
Instr
Data
25:21
20:16
15:0
A1
A2
A3
WE3
RD1
RD2
Register
File
WD3
A
SrcA
SrcB
4
00
01
10
11
ALU
ALUResult
ALUOut
SignImm
Sign Extend

Figure 7.23 Increment PC by 4

This completes the datapath for the `lw` instruction. Next, let us extend the datapath to also handle the `sw` instruction. Like the `lw` instruction, the `sw` instruction reads a base address from port 1 of the register file and sign-extends the immediate. The ALU adds the base address to the immediate to find the memory address. All of these functions are already supported by existing hardware in the datapath.

The only new feature of `sw` is that we must read a second register from the register file and write it into the memory, as shown in Figure 7.24. The register is specified in the `rt` field of the instruction, $Instr_{20:16}$, which is connected to the second port of the register file. When the register is read, it is stored in a nonarchitectural register, *B*. On the next step, it is sent to the write data port (*WD*) of the data memory to be written. The memory receives an additional *MemWrite* control signal to indicate that the write should occur.

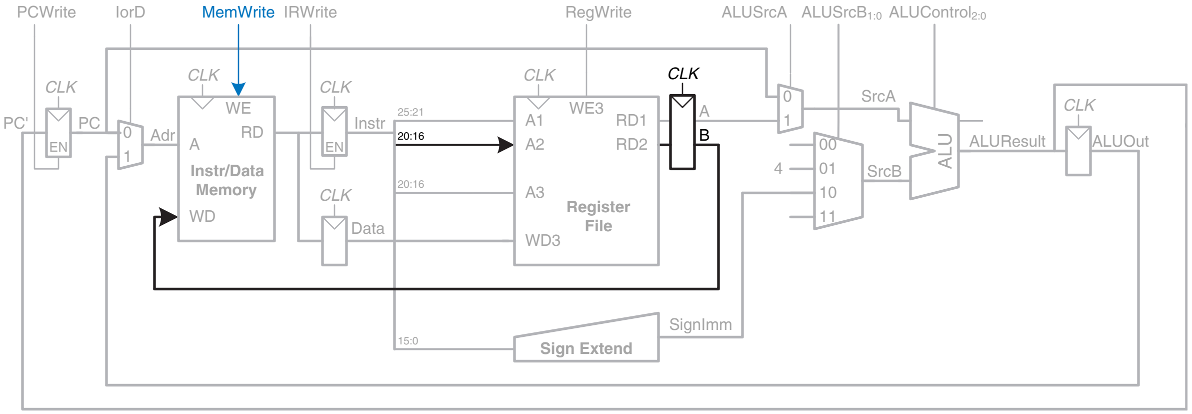

PCWrite
IorD
MemWrite
IRWrite
RegWrite
ALUSrcA
ALUSrcB1:0
ALUControl2:0
CLK
PC'
EN
PC
0
1
Adr
WE
A
RD
Instr/Data
Memory
WD
Instr
Data
25:21
20:16
20:16
15:0
A1
A2
A3
WE3
RD1
RD2
Register
File
WD3
A
B
SrcA
SrcB
4
00
01
10
11
ALU
ALUResult
ALUOut
SignImm
Sign Extend

Figure 7.24 Enhanced datapath for `sw` instruction

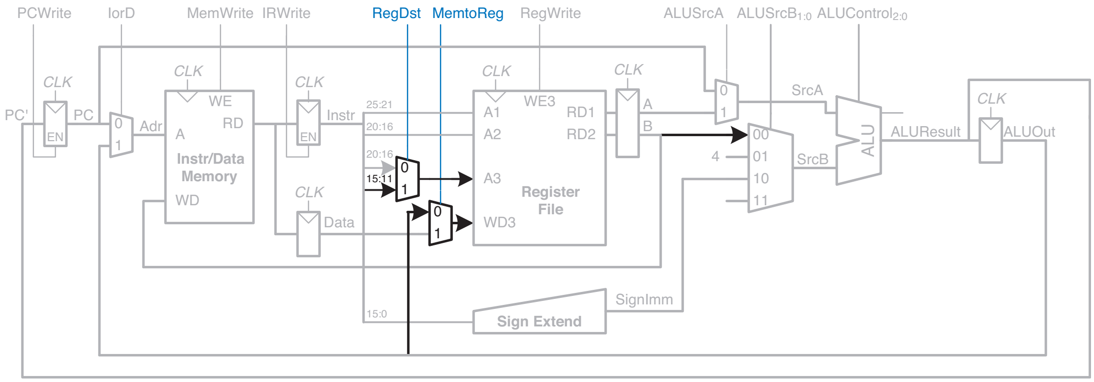

PCWrite
IorD
MemWrite
IRWrite
RegDst
MemtoReg
RegWrite
ALUSrcA
ALUSrcB1:0
ALUControl2:0
CLK
PC'
PC
EN
Adr
WE
RD
A
Instr/Data Memory
WD
Instr
Data
25:21
20:16
15:11
15:0
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
ALUResult
ALUOut
4
00
01
10
11
SignImm
Sign Extend

Figure 7.25 Enhanced datapath for R-type instructions

For R-type instructions, the instruction is again fetched, and the two source registers are read from the register file. $ALUSrcB_{1:0}$, the control input of the *SrcB* multiplexer, is used to choose register *B* as the second source register for the ALU, as shown in Figure 7.25. The ALU performs the appropriate operation and stores the result in *ALUOut*. On the next step, *ALUOut* is written back to the register specified by the `rd` field of the instruction, $Instr_{15:11}$. This requires two new multiplexers. The *MemtoReg* multiplexer selects whether *WD3* comes from *ALUOut* (for R-type instructions) or from *Data* (for `lw`). The *RegDst* instruction selects whether the destination register is specified in the `rt` or `rd` field of the instruction.

For the `beq` instruction, the instruction is again fetched, and the two source registers are read from the register file. To determine whether the registers are equal, the ALU subtracts the registers and, upon a zero result, sets the *Zero* flag. Meanwhile, the datapath must compute the next value of the PC if the branch is taken: $PC' = PC + 4 + SignImm \times 4$. In the single-cycle processor, yet another adder was needed to compute the branch address. In the multicycle processor, the ALU can be reused again to save hardware. On one step, the ALU computes $PC + 4$ and writes it back to the program counter, as was done for other instructions. On another step, the ALU uses this updated PC value to compute $PC + SignImm \times 4$. *SignImm* is left-shifted by 2 to multiply it by 4, as shown in Figure 7.26. The *SrcB* multiplexer chooses this value and adds it to the PC. This sum represents the destination of the branch and is stored in *ALUOut*. A new multiplexer, controlled by *PCSrc*, chooses what signal should be sent to *PC'*. The program counter should be written either when *PCWrite* is asserted or when a branch is taken. A new control signal, *Branch*, indicates that the `beq` instruction is being executed. The branch is taken if *Zero* is also asserted. Hence, the datapath computes a new PC write enable,

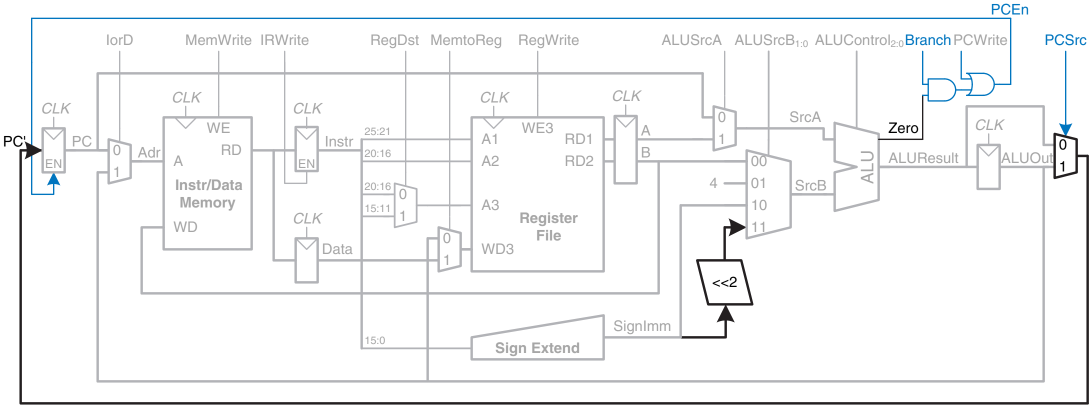

PCEn
IorD
MemWrite
IRWrite
RegDst
MemtoReg
RegWrite
ALUSrcA
ALUSrcB$_{1:0}$
ALUControl$_{2:0}$
Branch
PCWrite
PCSrc
CLK
PC'
PC
EN
0
1
Adr
A
WE
RD
Instr/Data Memory
WD
Instr
Data
25:21
20:16
15:11
15:0
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
A
B
SrcA
SrcB
4
00
01
10
11
<<2
SignImm
Sign Extend
ALU
Zero
ALUResult
ALUOut

Figure 7.26 Enhanced datapath for `beq` instruction

called *PCEn*, which is TRUE either when *PCWrite* is asserted or when both *Branch* and *Zero* are asserted.

This completes the design of the multicycle MIPS processor datapath. The design process is much like that of the single-cycle processor in that hardware is systematically connected between the state elements to handle each instruction. The main difference is that the instruction is executed in several steps. Nonarchitectural registers are inserted to hold the results of each step. In this way, the ALU can be reused several times, saving the cost of extra adders. Similarly, the instructions and data can be stored in one shared memory. In the next section, we develop an FSM controller to deliver the appropriate sequence of control signals to the datapath on each step of each instruction.

7.4.2 Multicycle Control

As in the single-cycle processor, the control unit computes the control signals based on the `opcode` and `funct` fields of the instruction, $Instr_{31:26}$ and $Instr_{5:0}$. Figure 7.27 shows the entire multicycle MIPS processor with the control unit attached to the datapath. The datapath is shown in black, and the control unit is shown in blue.

As in the single-cycle processor, the control unit is partitioned into a main controller and an ALU decoder, as shown in Figure 7.28. The ALU decoder is unchanged and follows the truth table of Table 7.2. Now, however, the main controller is an FSM that applies the proper control signals on the proper cycles or steps. The sequence of control signals depends on the instruction being executed. In the remainder of this section, we will develop the FSM state transition diagram for the main controller.

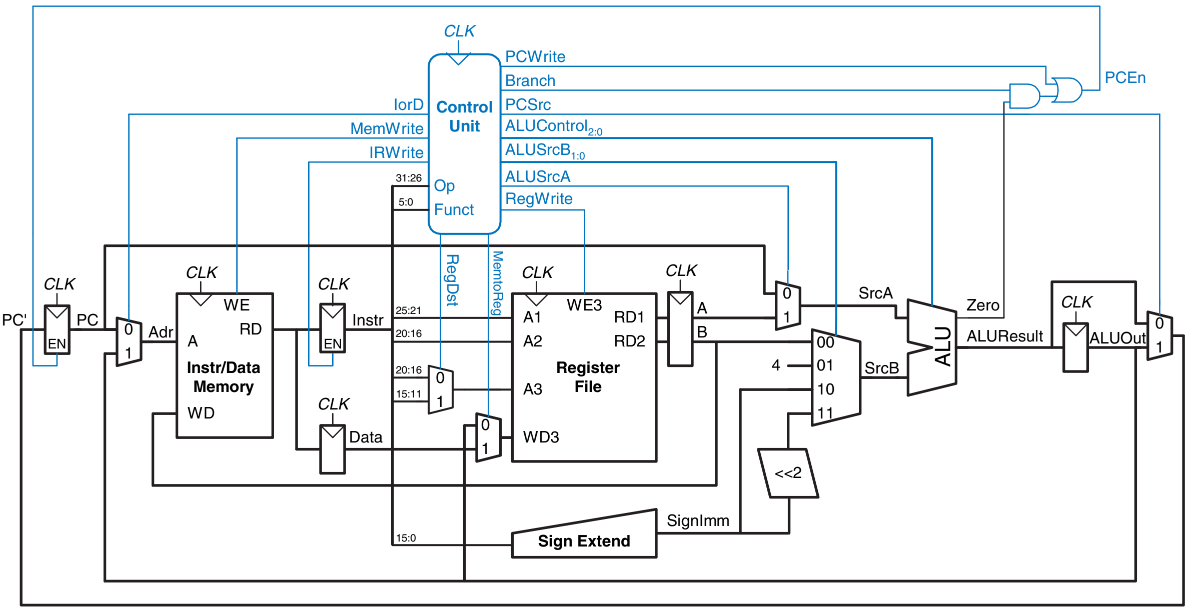

CLK
PCWrite
Branch
PCEn
IorD
MemWrite
IRWrite
Control Unit
PCSrc
ALUControl$_{2:0}$
ALUSrcB$_{1:0}$
ALUSrcA
RegWrite
31:26
5:0
Op
Funct
RegDst
MemtoReg
CLK
PC'
EN
PC
0
1
Adr
CLK
WE
RD
A
Instr/Data Memory
WD
CLK
EN
Instr
CLK
Data
25:21
20:16
20:16
15:11
15:0
CLK
WE3
A1
A2
A3
WD3
RD1
RD2
Register File
CLK
A
B
SrcA
00
01
10
11
4
SrcB
<<2
SignImm
Sign Extend
ALU
Zero
ALUResult
CLK
ALUOut

Figure 7.27 Complete multicycle MIPS processor

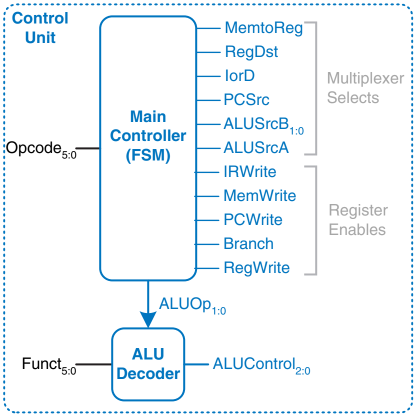

Control Unit
Opcode$_{5:0}$
Main Controller (FSM)
MemtoReg
RegDst
IorD
PCSrc
ALUSrcB$_{1:0}$
ALUSrcA
Multiplexer Selects
IRWrite
MemWrite
PCWrite
Branch
RegWrite
Register Enables
ALUOp$_{1:0}$
Funct$_{5:0}$
ALU Decoder
ALUControl$_{2:0}$

Figure 7.28 Control unit internal structure

The main controller produces multiplexer select and register enable signals for the datapath. The select signals are *MemtoReg*, *RegDst*, *IorD*, *PCSrc*, *ALUSrcB*, and *ALUSrcA*. The enable signals are *IRWrite*, *MemWrite*, *PCWrite*, *Branch*, and *RegWrite*.

To keep the following state transition diagrams readable, only the relevant control signals are listed. Select signals are listed only when their value matters; otherwise, they are don't cares. Enable signals are listed only when they are asserted; otherwise, they are 0.

The first step for any instruction is to fetch the instruction from memory at the address held in the PC. The FSM enters this state on reset. To read memory, $IorD = 0$, so the address is taken from the PC. $IRWrite$ is asserted to write the instruction into the instruction register, IR. Meanwhile, the PC should be incremented by 4 to point to the next instruction. Because the ALU is not being used for anything else, the processor can use it to compute $PC + 4$ at the same time that it fetches the instruction. $ALUSrcA = 0$, so $SrcA$ comes from the PC. $ALUSrcB = 01$, so $SrcB$ is the constant 4. $ALUOp = 00$, so the ALU decoder produces $ALUControl = 010$ to make the ALU add. To update the PC with this new value, $PCSrc = 0$, and $PCWrite$ is asserted. These control signals are shown in Figure 7.29. The data flow on this step is shown in Figure 7.30, with the instruction fetch shown using the dashed blue line and the PC increment shown using the dashed gray line.

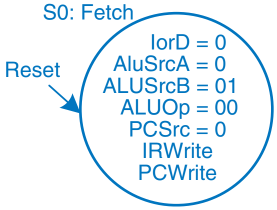

S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite

Figure 7.29 **Fetch**

The next step is to read the register file and decode the instruction. The register file always reads the two sources specified by the `rs` and `rt` fields of the instruction. Meanwhile, the immediate is sign-extended. Decoding involves examining the `opcode` of the instruction to determine what to do next. No control signals are necessary to decode the instruction, but the FSM must wait 1 cycle for the reading and decoding to complete, as shown in Figure 7.31. The new state is highlighted in blue. The data flow is shown in Figure 7.32.

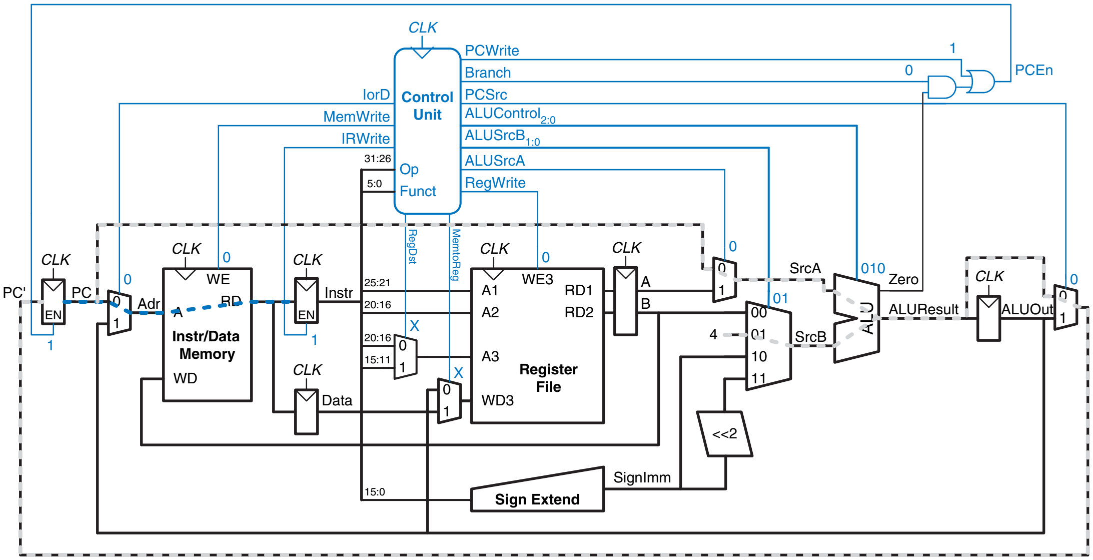

CLK
Control Unit
PCWrite
Branch
PCSrc
ALUControl2:0
ALUSrcB1:0
ALUSrcA
RegWrite
IorD
MemWrite
IRWrite
31:26
5:0
Op
Funct
1
0
PCEn
RegDst
MemtoReg
PC'
PC
EN
Adr
A
RD
WE
WD
Instr/Data Memory
Instr
Data
25:21
20:16
15:11
15:0
X
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
4
00
01
10
11
<<2
SignImm
Sign Extend
010
ALU
Zero
ALUResult
ALUOut

Figure 7.30 **Data flow during the fetch step**

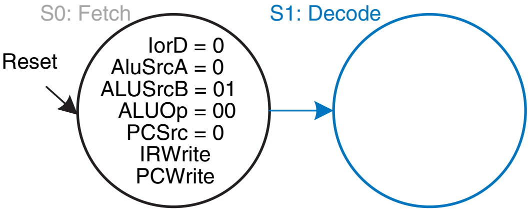

S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
S1: Decode

Figure 7.31 **Decode**

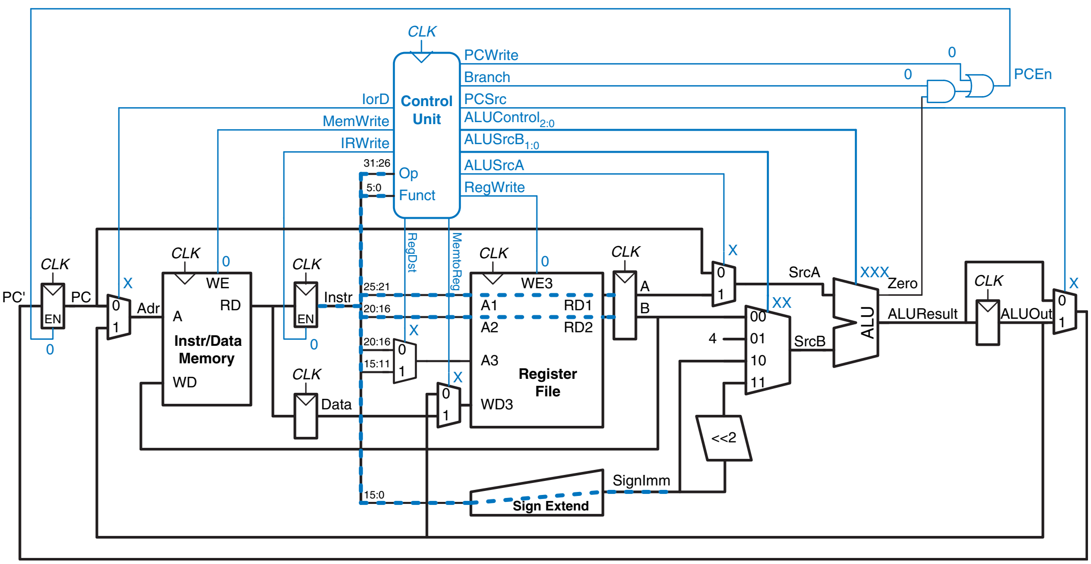

CLK
Control Unit
PCWrite
Branch
PCSrc
ALUControl$_{2:0}$
ALUSrcB$_{1:0}$
ALUSrcA
RegWrite
IorD
MemWrite
IRWrite
31:26
Op
5:0
Funct
RegDst
MemtoReg
PCEn
0
0
PC'
PC
EN
0
1
X
Adr
A
WE
RD
Instr/Data Memory
WD
Instr
Data
25:21
20:16
20:16
15:11
15:0
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
A
B
SrcA
SrcB
4
00
01
10
11
XX
XXX
ALU
Zero
ALUResult
ALUOut
<<2
Sign Extend
SignImm

Figure 7.32 **Data flow during the decode step**

Now the FSM proceeds to one of several possible states, depending on the `opcode`. If the instruction is a memory load or store (`lw` or `sw`), the multicycle processor computes the address by adding the base address to the sign-extended immediate. This requires $ALUSrcA = 1$ to select register A and $ALUSrcB = 10$ to select $SignImm$. $ALUOp = 00$, so the ALU adds. The effective address is stored in the $ALUOut$ register for use on the next step. This FSM step is shown in Figure 7.33, and the data flow is shown in Figure 7.34.

If the instruction is `lw`, the multicycle processor must next read data from memory and write it to the register file. These two steps are shown in Figure 7.35. To read from memory, $IorD = 1$ to select the memory address that was just computed and saved in $ALUOut$. This address in memory is read and saved in the Data register during step S3. On the next step, S4, $Data$ is written to the register file. $MemtoReg = 1$ to select

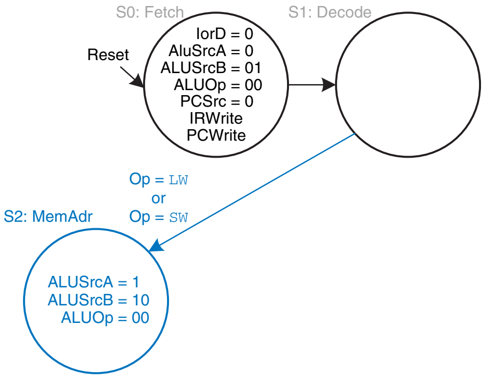

S0: Fetch
S1: Decode
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
Op = LW
or
Op = SW
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

Figure 7.33 **Memory address computation**

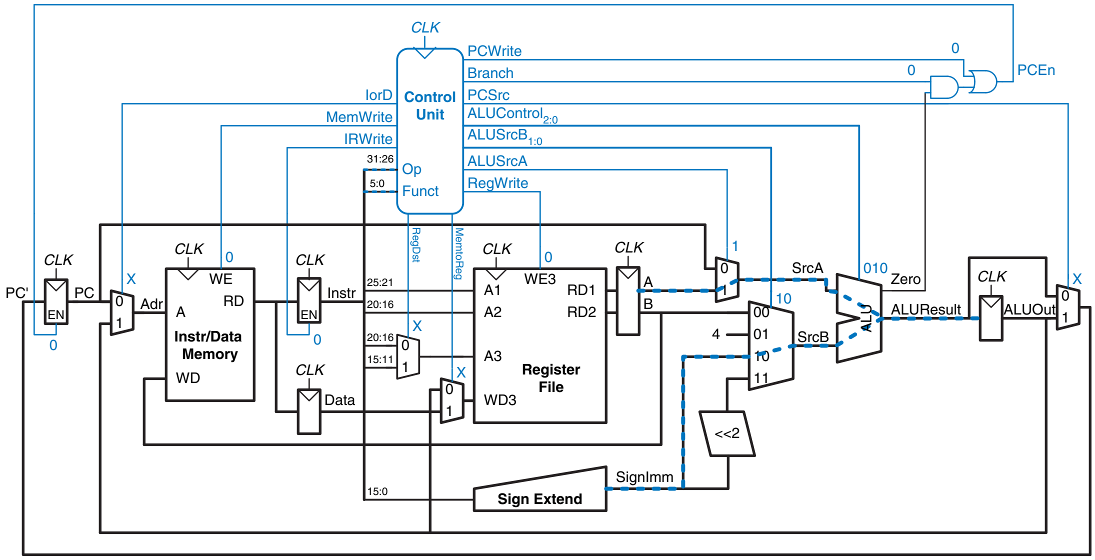

CLK
Control Unit
PCWrite
Branch
PCSrc
ALUControl$_{2:0}$
ALUSrcB$_{1:0}$
ALUSrcA
RegWrite
IorD
MemWrite
IRWrite
31:26
5:0
Op
Funct
RegDst
MemtoReg
PCEn
0
0
PC'
PC
EN
X
0
Adr
WE
RD
A
Instr/Data Memory
WD
Instr
Data
25:21
20:16
20:16
15:11
15:0
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
A
B
SrcA
SrcB
4
00
01
10
11
<<2
SignImm
Sign Extend
ALU
010
Zero
ALUResult
ALUOut
1

Figure 7.34 **Data flow during memory address computation**

Data, and *RegDst* = 0 to pull the destination register from the `rt` field of the instruction. *RegWrite* is asserted to perform the write, completing the `lw` instruction. Finally, the FSM returns to the initial state, S0, to fetch the next instruction. For these and subsequent steps, try to visualize the data flow on your own.

From state S2, if the instruction is `sw`, the data read from the second port of the register file is simply written to memory. In state S3, *IorD* = 1 to select the address computed in S2 and saved in *ALUOut*. *MemWrite* is asserted to write the memory. Again, the FSM returns to S0 to fetch the next instruction. The added step is shown in Figure 7.36.

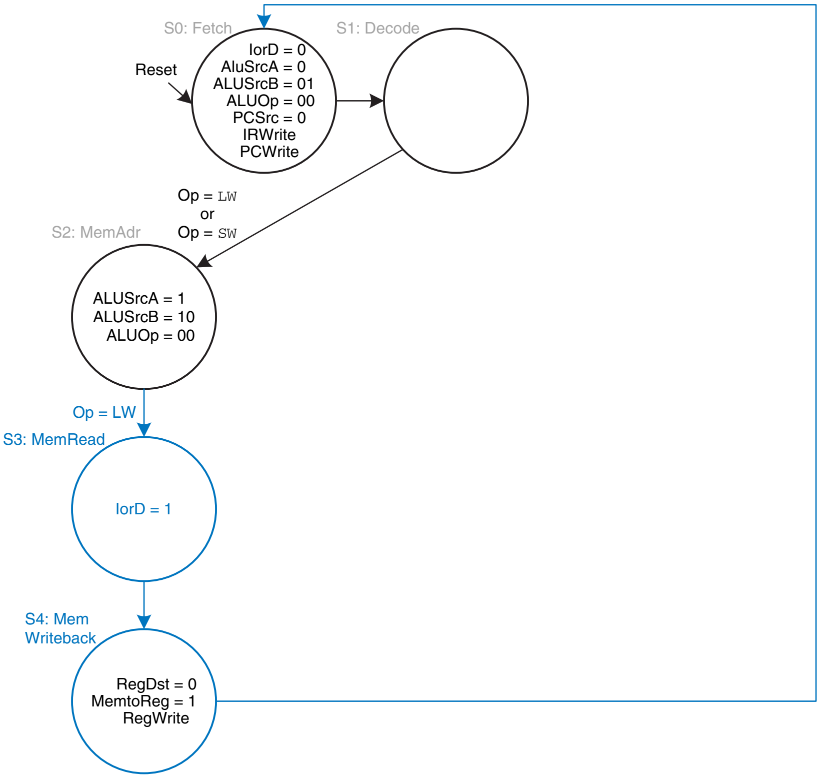

S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
S1: Decode
Op = LW
or
Op = SW
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Op = LW
S3: MemRead
IorD = 1
S4: Mem
Writeback
RegDst = 0
MemtoReg = 1
RegWrite

Figure 7.35 **Memory read**

If the `opcode` indicates an R-type instruction, the multicycle processor must calculate the result using the ALU and write that result to the register file. Figure 7.37 shows these two steps. In S6, the instruction is executed by selecting the A and B registers ($ALUSrcA = 1$, $ALUSrcB = 00$) and performing the ALU operation indicated by the `funct` field of the instruction. $ALUOp = 10$ for all R-type instructions. The *ALUResult* is stored in *ALUOut*. In S7, *ALUOut* is written to the register file, $RegDst = 1$, because the destination register is specified in the `rd` field of the instruction. $MemtoReg = 0$ because the write data, *WD3*, comes from *ALUOut*. *RegWrite* is asserted to write the register file.

For a `beq` instruction, the processor must calculate the destination address and compare the two source registers to determine whether the branch should be taken. This requires two uses of the ALU and hence might seem to demand two new states. Notice, however, that the ALU was not used during S1 when the registers were being read. The processor might as well use the ALU at that time to compute the destination address by adding the incremented PC, $PC + 4$, to $SignImm \times 4$, as shown in Figure 7.38 (see page 404). $ALUSrcA = 0$ to select the incremented PC,

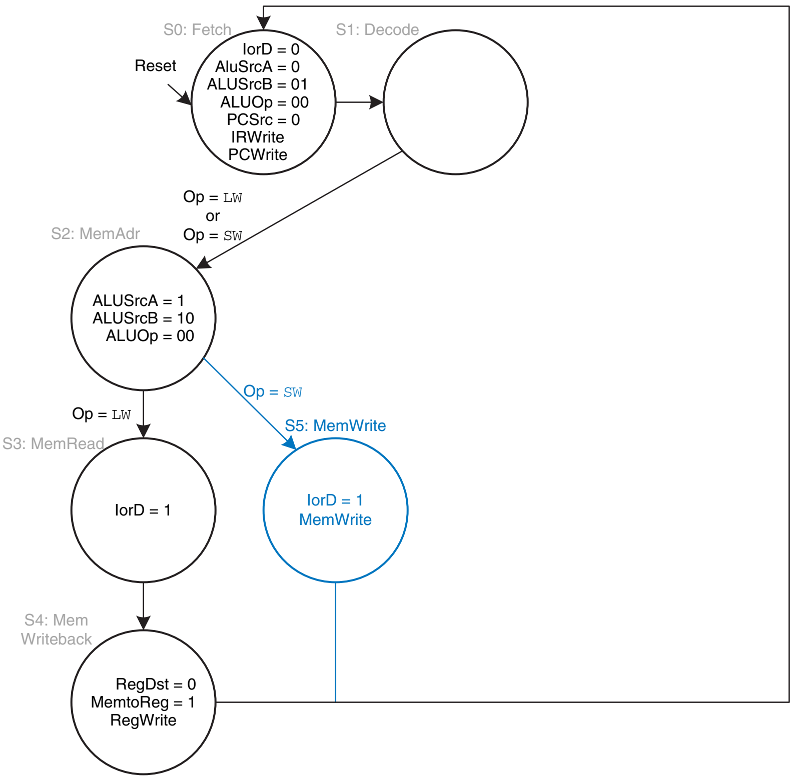

S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
S1: Decode
Op = LW
or
Op = SW
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Op = LW
Op = SW
S3: MemRead
IorD = 1
S5: MemWrite
IorD = 1
MemWrite
S4: Mem
Writeback
RegDst = 0
MemtoReg = 1
RegWrite

Figure 7.36 **Memory write**

ALUSrcB = 11 to select *SignImm* × 4, and *ALUOp* = 00 to add. The destination address is stored in *ALUOut*. If the instruction is not `beq`, the computed address will not be used in subsequent cycles, but its computation was harmless. In S8, the processor compares the two registers by subtracting them and checking to determine whether the result is 0. If it is, the processor branches to the address that was just computed. *ALUSrcA* = 1 to select register *A*; *ALUSrcB* = 00 to select register *B*; *ALUOp* = 01 to subtract; *PCSrc* = 1 to take the destination address from *ALUOut*, and *Branch* = 1 to update the PC with this address if the ALU result is 0.[2]

Putting these steps together, Figure 7.39 shows the complete main controller state transition diagram for the multicycle processor (see page 405). Converting it to hardware is a straightforward but tedious task using the techniques of Chapter 3. Better yet, the FSM can be coded in an HDL and synthesized using the techniques of Chapter 4.

---

[2] Now we see why the *PCSrc* multiplexer is necessary to choose *PC′* from either *ALUResult* (in S0) or *ALUOut* (in S8).

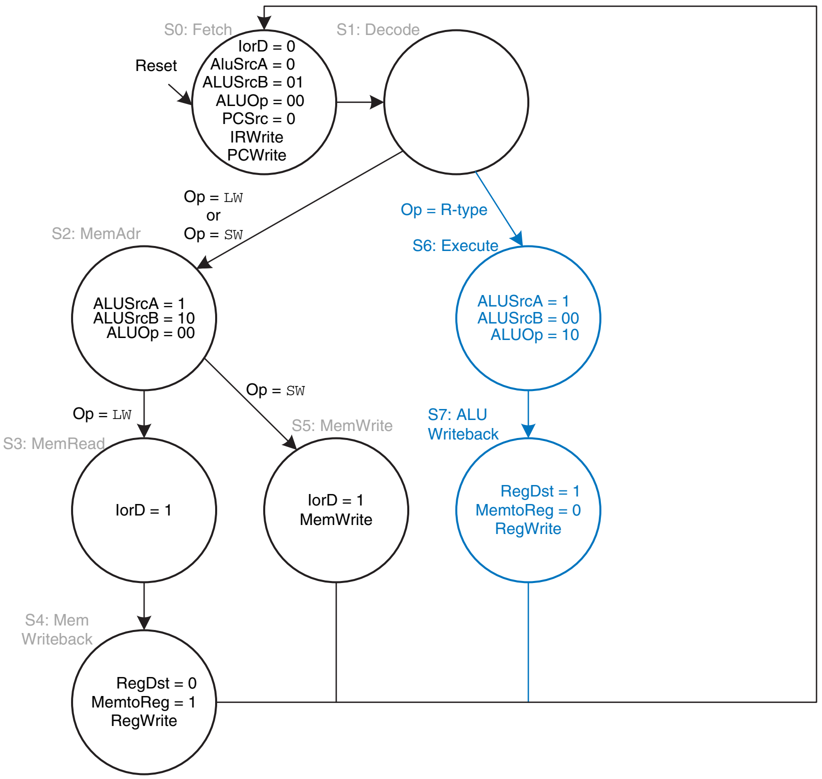

S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
S1: Decode
Op = LW
or
Op = SW
Op = R-type
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
S6: Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Op = SW
Op = LW
S3: MemRead
IorD = 1
S5: MemWrite
IorD = 1
MemWrite
S7: ALU
Writeback
RegDst = 1
MemtoReg = 0
RegWrite
S4: Mem
Writeback
RegDst = 0
MemtoReg = 1
RegWrite

Figure 7.37 **Execute R-type operation**

7.4.3 More Instructions

As we did in Section 7.3.3 for the single-cycle processor, let us now extend the multicycle processor to support the `addi` and `j` instructions. The next two examples illustrate the general design process to support new instructions.

Example 7.5 `addi` INSTRUCTION

Modify the multicycle processor to support `addi`.

Solution: The datapath is already capable of adding registers to immediates, so all we need to do is add new states to the main controller FSM for `addi`, as shown in Figure 7.40 (see page 406). The states are similar to those for R-type instructions. In S9, register *A* is added to *SignImm* (*ALUSrcA* = 1, *ALUSrcB* = 10, *ALUOp* = 00) and the result, *ALUResult*, is stored in *ALUOut*. In S10, *ALUOut* is written

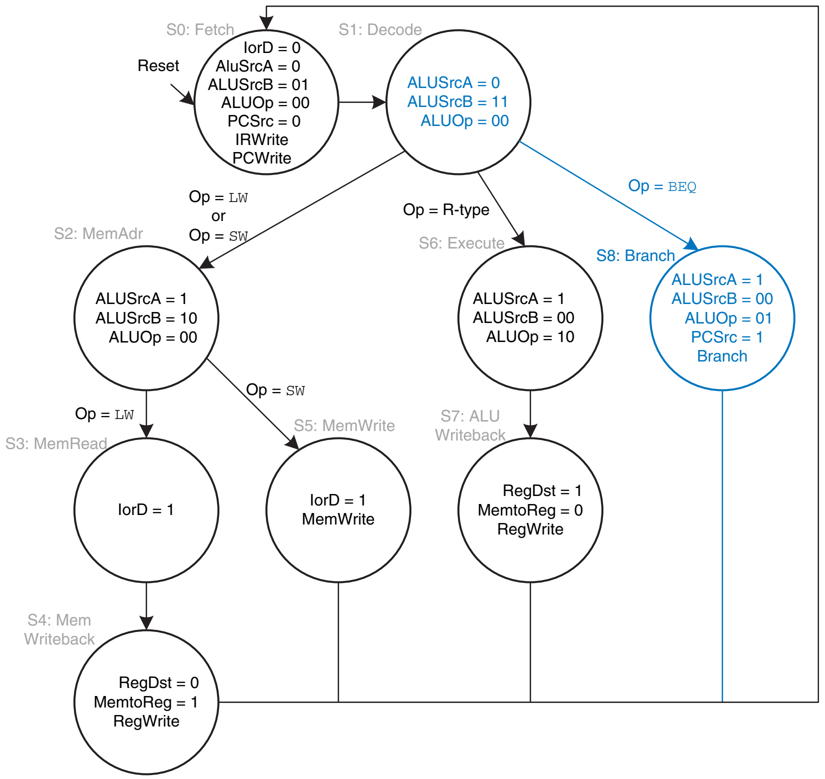

S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
S1: Decode
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
Op = LW
or
Op = SW
Op = R-type
Op = BEQ
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
S6: Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
S8: Branch
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch
Op = LW
Op = SW
S3: MemRead
IorD = 1
S5: MemWrite
IorD = 1
MemWrite
S7: ALU Writeback
RegDst = 1
MemtoReg = 0
RegWrite
S4: Mem Writeback
RegDst = 0
MemtoReg = 1
RegWrite

Figure 7.38 **Branch**

to the register specified by the `rt` field of the instruction (*RegDst* = 0, *MemtoReg* = 0, *RegWrite* asserted). The astute reader may notice that S2 and S9 are identical and could be merged into a single state.

Example 7.6 `j` INSTRUCTION

Modify the multicycle processor to support `j`.

Solution: First, we must modify the datapath to compute the next PC value in the case of a `j` instruction. Then we add a state to the main controller to handle the instruction.

Figure 7.41 shows the enhanced datapath (see page 407). The jump destination address is formed by left-shifting the 26-bit `addr` field of the instruction by two bits, then prepending the four most significant bits of the already incremented PC. The *PCSrc* multiplexer is extended to take this address as a third input.

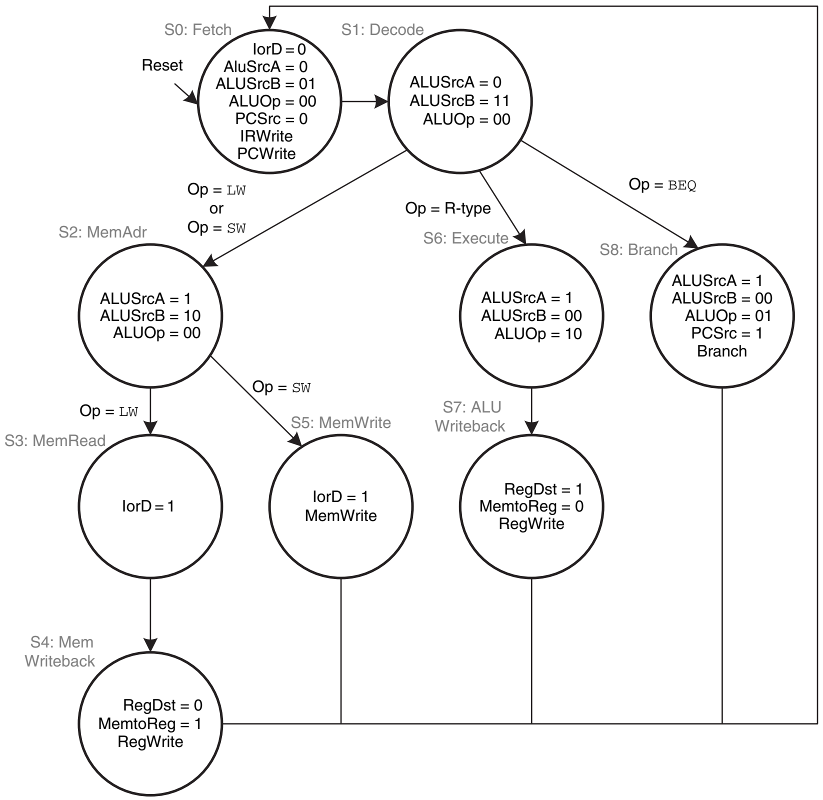

S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
S1: Decode
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
Op = LW
or
Op = SW
Op = R-type
Op = BEQ
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
S6: Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
S8: Branch
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch
Op = LW
Op = SW
S3: MemRead
IorD = 1
S5: MemWrite
IorD = 1
MemWrite
S7: ALU
Writeback
RegDst = 1
MemtoReg = 0
RegWrite
S4: Mem
Writeback
RegDst = 0
MemtoReg = 1
RegWrite

Figure 7.39 **Complete multicycle control FSM**

Figure 7.42 shows the enhanced main controller (see page 408). The new state, S11, simply selects *PC′* as the *PCJump* value (*PCSrc* = 10) and writes the PC. Note that the *PCSrc* select signal is extended to two bits in S0 and S8 as well.

7.4.4 Performance Analysis

The execution time of an instruction depends on both the number of cycles it uses and the cycle time. Whereas the single-cycle processor performed all instructions in one cycle, the multicycle processor uses varying numbers of cycles for the various instructions. However, the multicycle processor does less work in a single cycle and, thus, has a shorter cycle time.

The multicycle processor requires three cycles for `beq` and `j` instructions, four cycles for `sw`, `addi`, and R-type instructions, and five cycles for `lw` instructions. The CPI depends on the relative likelihood that each instruction is used.

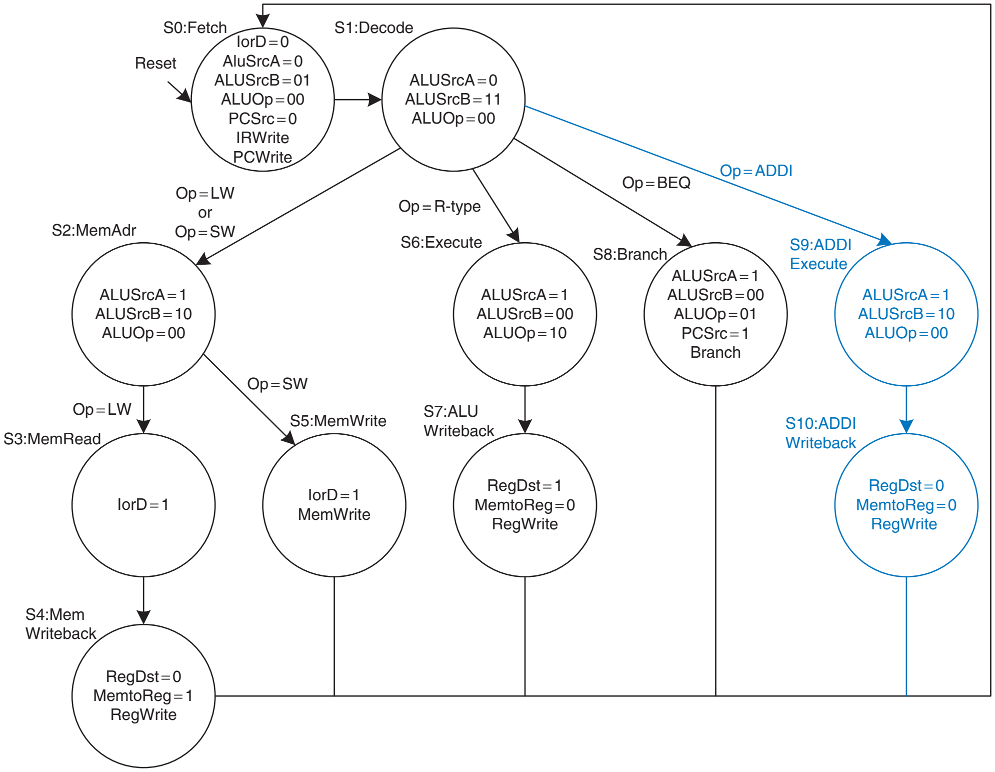

S0:Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 0
IRWrite
PCWrite
S1:Decode
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
Op = LW
or
Op = SW
S2:MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Op = R-type
S6:Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Op = BEQ
S8:Branch
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 1
Branch
Op = ADDI
S9:ADDI
Execute
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Op = SW
Op = LW
S3:MemRead
IorD = 1
S5:MemWrite
IorD = 1
MemWrite
S7:ALU
Writeback
RegDst = 1
MemtoReg = 0
RegWrite
S10:ADDI
Writeback
RegDst = 0
MemtoReg = 0
RegWrite
S4:Mem
Writeback
RegDst = 0
MemtoReg = 1
RegWrite

Figure 7.40 **Main controller states for** `addi`

Example 7.7 MULTICYCLE PROCESSOR CPI

The SPECINT2000 benchmark consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions.[3] Determine the average CPI for this benchmark.

Solution: The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of the time that instruction is used. For this benchmark, Average CPI $= (0.11 + 0.02)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12$. This is better than the worst-case CPI of 5, which would be required if all instructions took the same time.

---

[3] Data from Patterson and Hennessy, *Computer Organization and Design*, 4th Edition, Morgan Kaufmann, 2011.

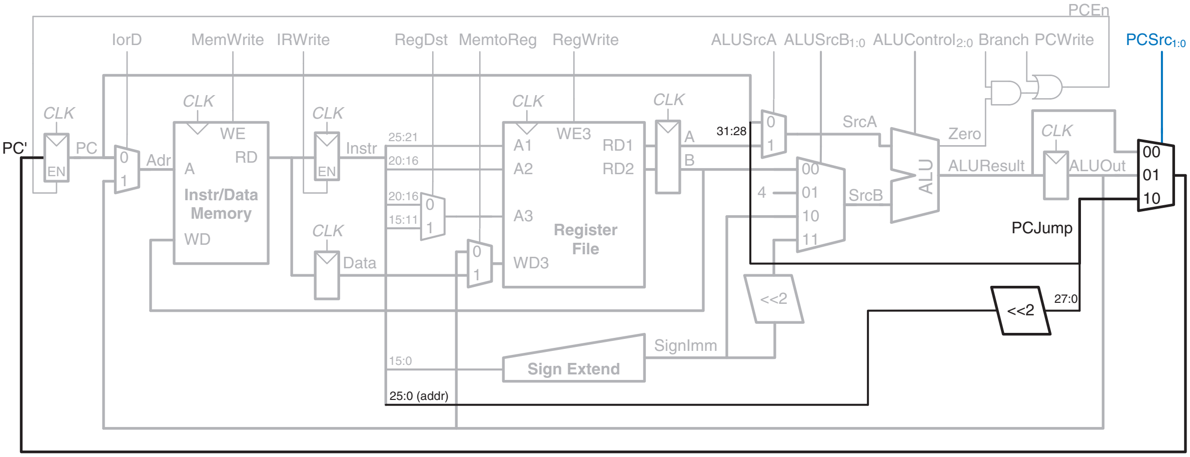

PCEn
IorD
MemWrite
IRWrite
RegDst
MemtoReg
RegWrite
ALUSrcA
$ALUSrcB_{1:0}$
$ALUControl_{2:0}$
Branch
PCWrite
$PCSrc_{1:0}$
CLK
PC'
PC
EN
0
1
Adr
WE
RD
A
Instr/Data Memory
WD
Instr
25:21
20:16
15:11
Data
A1
A2
A3
WE3
RD1
RD2
WD3
Register File
A
B
31:28
SrcA
SrcB
4
00
01
10
11
ALU
Zero
ALUResult
ALUOut
PCJump
<<2
27:0
SignImm
Sign Extend
15:0
25:0 (addr)

Figure 7.41 Multicycle MIPS datapath enhanced to support the `j` instruction

Recall that we designed the multicycle processor so that each cycle involved one ALU operation, memory access, or register file access. Let us assume that the register file is faster than the memory and that writing memory is faster than reading memory. Examining the datapath reveals two possible critical paths that would limit the cycle time:

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux},\ t_{mem}) + t_{\text{setup}} \tag{7.4}$$

The numerical values of these times will depend on the specific implementation technology.

Example 7.8 PROCESSOR PERFORMANCE COMPARISON

Ben Bitdiddle is wondering whether he would be better off building the multicycle processor instead of the single-cycle processor. For both designs, he plans on using a 65 nm CMOS manufacturing process with the delays given in Table 7.6. Help him compare each processor's execution time for 100 billion instructions from the SPECINT2000 benchmark (see Example 7.7).

Solution: According to Equation 7.4, the cycle time of the multicycle processor is $T_{c2} = 30 + 25 + 250 + 20 = 325$ ps. Using the CPI of 4.12 from Example 7.7, the total execution time is $T_2 = (100 \times 10^9$ instructions)(4.12 cycles/instruction) $(325 \times 10^{-12}$ s/cycle) = 133.9 seconds. According to Example 7.4, the single-cycle processor had a cycle time of $T_{c1} = 925$ ps, a CPI of 1, and a total execution time of 92.5 seconds.

One of the original motivations for building a multicycle processor was to avoid making all instructions take as long as the slowest one. Unfortunately, this

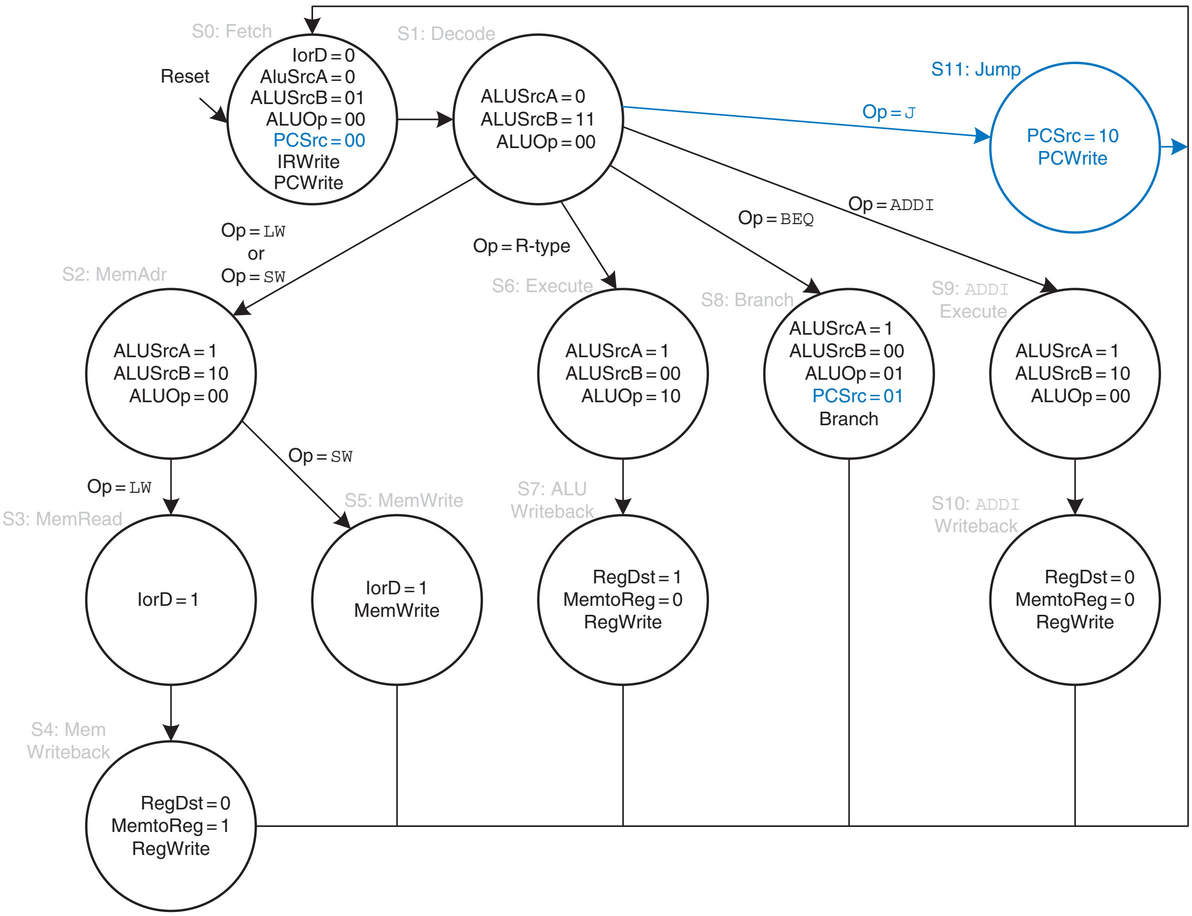

S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 00
IRWrite
PCWrite
S1: Decode
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
S11: Jump
Op = J
PCSrc = 10
PCWrite
Op = ADDI
Op = BEQ
Op = LW
or
Op = SW
Op = R-type
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
S6: Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
S8: Branch
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 01
Branch
S9: ADDI
Execute
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Op = SW
Op = LW
S3: MemRead
IorD = 1
S5: MemWrite
IorD = 1
MemWrite
S7: ALU
Writeback
RegDst = 1
MemtoReg = 0
RegWrite
S10: ADDI
Writeback
RegDst = 0
MemtoReg = 0
RegWrite
S4: Mem
Writeback
RegDst = 0
MemtoReg = 1
RegWrite

Figure 7.42 **Main controller state for `j`**

example shows that the multicycle processor is slower than the single-cycle processor given the assumptions of CPI and circuit element delays. The fundamental problem is that even though the slowest instruction, `lw`, was broken into five steps, the multicycle processor cycle time was not nearly improved five-fold. This is partly because not all of the steps are exactly the same length, and partly because the 50-ps sequencing overhead of the register clk-to-Q and setup time must now be paid on every step, not just once for the entire instruction. In general, engineers have learned that it is difficult to exploit the fact that some computations are faster than others unless the differences are large.

Compared with the single-cycle processor, the multicycle processor is likely to be less expensive because it eliminates two adders and combines the instruction and data memories into a single unit. It does, however, require five nonarchitectural registers and additional multiplexers.

7.5 PIPELINED PROCESSOR

Pipelining, introduced in Section 3.6, is a powerful way to improve the throughput of a digital system. We design a pipelined processor by subdividing the single-cycle processor into five pipeline stages. Thus, five instructions can execute simultaneously, one in each stage. Because each stage has only one-fifth of the entire logic, the clock frequency is almost five times faster. Hence, the latency of each instruction is ideally unchanged, but the throughput is ideally five times better. Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency. Pipelining introduces some overhead, so the throughput will not be quite as high as we might ideally desire, but pipelining nevertheless gives such great advantage for so little cost that all modern high-performance microprocessors are pipelined.

Reading and writing the memory and register file and using the ALU typically constitute the biggest delays in the processor. We choose five pipeline stages so that each stage involves exactly one of these slow steps. Specifically, we call the five stages *Fetch*, *Decode*, *Execute*, *Memory*, and *Writeback*. They are similar to the five steps that the multicycle processor used to perform `lw`. In the *Fetch* stage, the processor reads the instruction from instruction memory. In the *Decode* stage, the processor reads the source operands from the register file and decodes the instruction to produce the control signals. In the *Execute* stage, the processor performs a computation with the ALU. In the *Memory* stage, the processor reads or writes data memory. Finally, in the *Writeback* stage, the processor writes the result to the register file, when applicable.

Figure 7.43 shows a timing diagram comparing the single-cycle and pipelined processors. Time is on the horizontal axis, and instructions are on the vertical axis. The diagram assumes the logic element delays from Table 7.6 but ignores the delays of multiplexers and registers. In the single-cycle processor, Figure 7.43(a), the first instruction is read from memory at time 0; next the operands are read from the register file; and then the ALU executes the necessary computation. Finally, the data memory may be accessed, and the result is written back to the register file by 950 ps. The second instruction begins when the first completes. Hence, in this diagram, the single-cycle processor has an instruction latency of $250 + 150 + 200 + 250 + 100 = 950$ ps and a throughput of 1 instruction per 950 ps (1.05 billion instructions per second).

In the pipelined processor, Figure 7.43(b), the length of a pipeline stage is set at 250 ps by the slowest stage, the memory access (in the Fetch or Memory stage). At time 0, the first instruction is fetched from memory. At 250 ps, the first instruction enters the Decode stage, and a

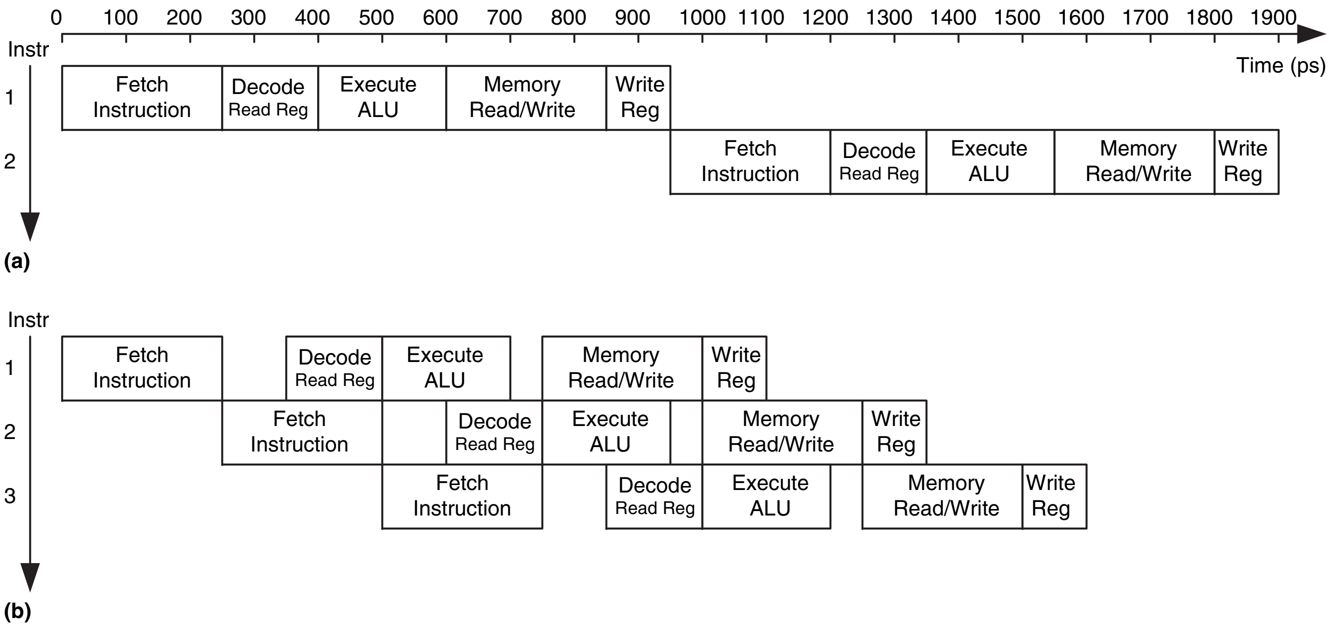

0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
Time (ps)
Instr
1
2
Fetch Instruction
Decode Read Reg
Execute ALU
Memory Read/Write
Write Reg
(a)
Instr
1
2
3
Fetch Instruction
Decode Read Reg
Execute ALU
Memory Read/Write
Write Reg
(b)

Figure 7.43 Timing diagrams: (a) single-cycle processor, (b) pipelined processor

second instruction is fetched. At 500 ps, the first instruction executes, the second instruction enters the Decode stage, and a third instruction is fetched. And so forth, until all the instructions complete. The instruction latency is $5 \times 250 = 1250$ ps. The throughput is 1 instruction per 250 ps (4 billion instructions per second). Because the stages are not perfectly balanced with equal amounts of logic, the latency is slightly longer for the pipelined than for the single-cycle processor. Similarly, the throughput is not quite five times as great for a five-stage pipeline as for the single-cycle processor. Nevertheless, the throughput advantage is substantial.

Figure 7.44 shows an abstracted view of the pipeline in operation in which each stage is represented pictorially. Each pipeline stage is represented with its major component—instruction memory (IM), register file (RF) read, ALU execution, data memory (DM), and register file writeback—to illustrate the flow of instructions through the pipeline. Reading across a row shows the clock cycles in which a particular instruction is in each stage. For example, the `sub` instruction is fetched in cycle 3 and executed in cycle 5. Reading down a column shows what the various pipeline stages are doing on a particular cycle. For example, in cycle 6, the `or` instruction is being fetched from instruction memory, while `$s1` is being read from the register file, the ALU is computing `$t5` AND `$t6`, the data memory is idle, and the register file is writing a sum to `$s3`. Stages are shaded to indicate when they are used. For example, the data memory is used by `lw` in cycle 4 and by `sw` in cycle 8. The instruction memory and ALU are used in every cycle. The register file is written by every

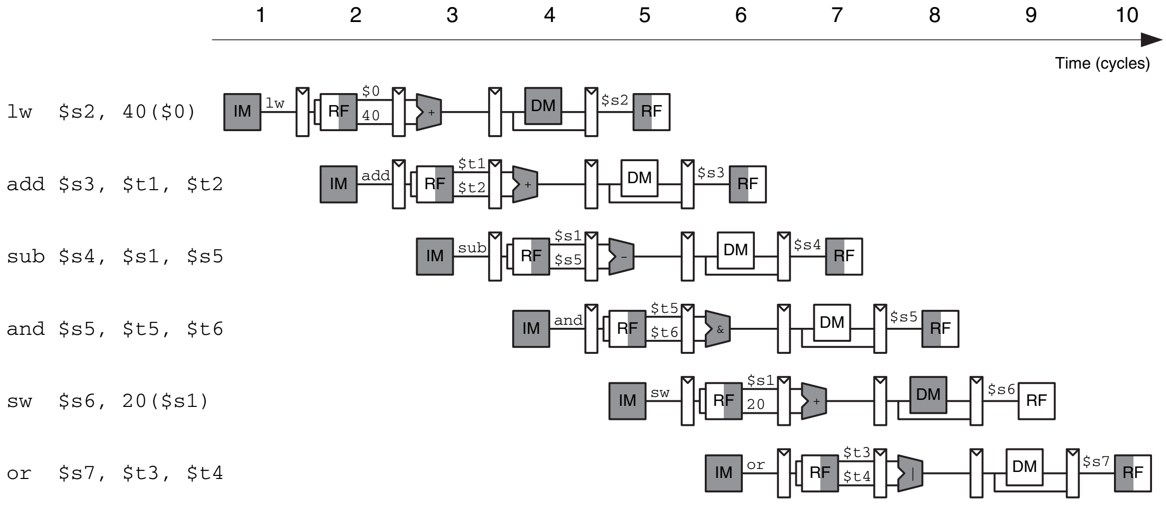

1
2
3
4
5
6
7
8
9
10
Time (cycles)
lw $s2, 40($0)
IM
lw
RF
$0
40
+
DM
$s2
RF
add $s3, $t1, $t2
IM
add
RF
$t1
$t2
+
DM
$s3
RF
sub $s4, $s1, $s5
IM
sub
RF
$s1
$s5
-
DM
$s4
RF
and $s5, $t5, $t6
IM
and
RF
$t5
$t6
&
DM
$s5
RF
sw $s6, 20($s1)
IM
sw
RF
$s1
20
+
DM
$s6
RF
or $s7, $t3, $t4
IM
or
RF
$t3
$t4
|
DM
$s7
RF

Figure 7.44 Abstract view of pipeline in operation

instruction except `sw`. In the pipelined processor, the register file is written in the first part of a cycle and read in the second part, as suggested by the shading. This way, data can be written and read back within a single cycle.

A central challenge in pipelined systems is handling *hazards* that occur when the results of one instruction are needed by a subsequent instruction before the former instruction has completed. For example, if the `add` in Figure 7.44 used `$s2` rather than `$t2`, a hazard would occur because the `$s2` register has not been written by the `lw` by the time it is read by the `add`. This section explores *forwarding*, *stalls*, and *flushes* as methods to resolve hazards. Finally, this section revisits performance analysis considering sequencing overhead and the impact of hazards.

7.5.1 Pipelined Datapath

The pipelined datapath is formed by chopping the single-cycle datapath into five stages separated by pipeline registers. Figure 7.45(a) shows the single-cycle datapath stretched out to leave room for the pipeline registers. Figure 7.45(b) shows the pipelined datapath formed by inserting four pipeline registers to separate the datapath into five stages. The stages and their boundaries are indicated in blue. Signals are given a suffix (F, D, E, M, or W) to indicate the stage in which they reside.

The register file is peculiar because it is read in the Decode stage and written in the Writeback stage. It is drawn in the Decode stage, but the write address and data come from the Writeback stage. This feedback will lead to pipeline hazards, which are discussed in Section 7.5.3. The register file in the pipelined processor writes on the falling edge of *CLK*, when *WD3* is stable.

One of the subtle but critical issues in pipelining is that all signals associated with a particular instruction must advance through the pipeline in unison. Figure 7.45(b) has an error related to this issue. Can you find it?

The error is in the register file write logic, which should operate in the Writeback stage. The data value comes from *ResultW*, a Writeback stage signal. But the address comes from *WriteRegE*, an Execute stage signal. In the pipeline diagram of Figure 7.44, during cycle 5, the result of the `lw` instruction would be incorrectly written to `$s4` rather than `$s2`.

Figure 7.46 shows a corrected datapath. The *WriteReg* signal is now pipelined along through the Memory and Writeback stages, so it remains in sync with the rest of the instruction. *WriteRegW* and *ResultW* are fed back together to the register file in the Writeback stage.

The astute reader may notice that the *PC′* logic is also problematic, because it might be updated with a Fetch or a Memory stage signal (*PCPlus4F* or *PCBranchM*). This control hazard will be fixed in Section 7.5.3.

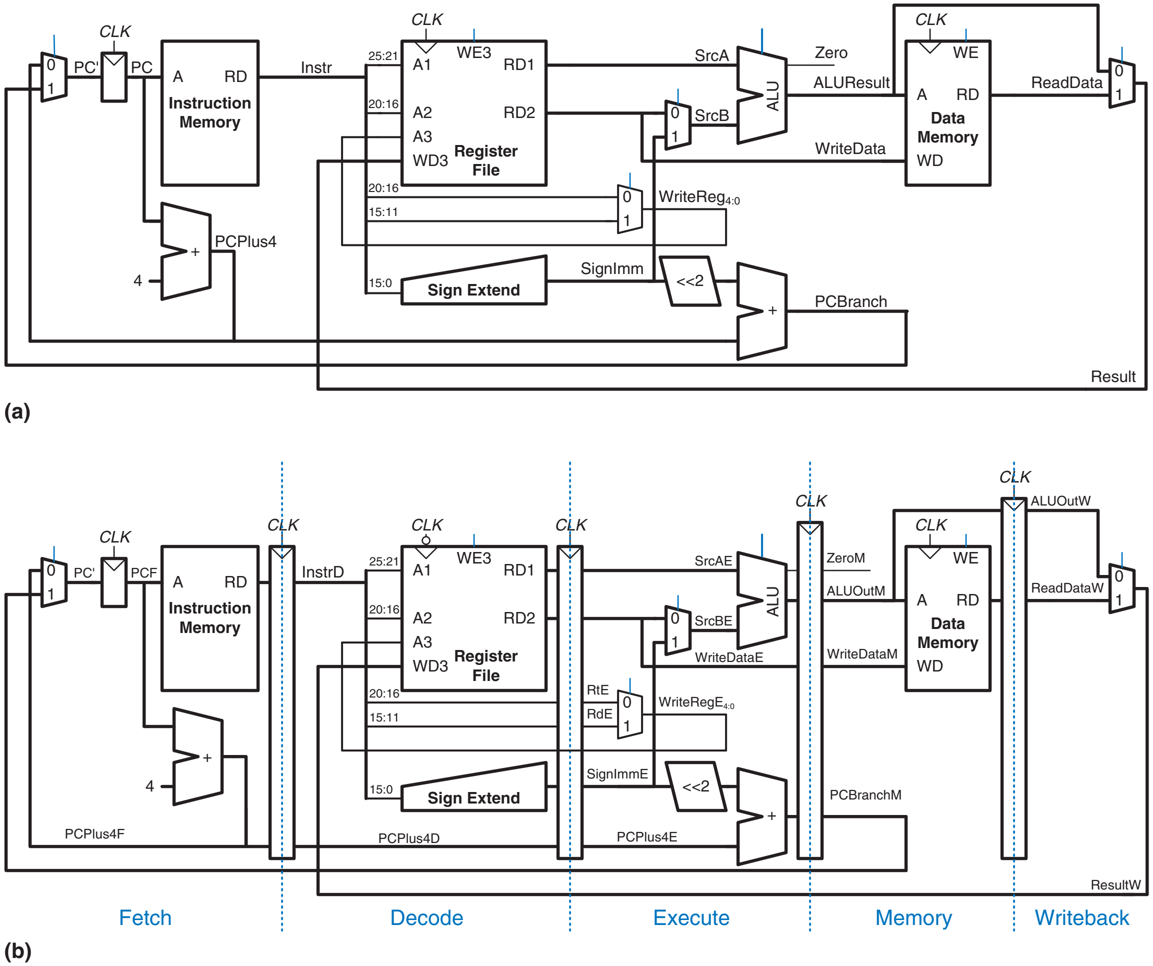

CLK
PC'
PC
A
RD
Instruction Memory
Instr
25:21
20:16
15:11
15:0
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
Zero
ALUResult
WriteData
WE
Data Memory
ReadData
WriteReg4:0
PCPlus4
4
Sign Extend
SignImm
<<2
PCBranch
Result
(a)
InstrD
PCF
SrcAE
SrcBE
WriteDataE
RtE
RdE
WriteRegE4:0
SignImmE
PCPlus4F
PCPlus4D
PCPlus4E
ZeroM
ALUOutM
WriteDataM
PCBranchM
ALUOutW
ReadDataW
ResultW
Fetch
Decode
Execute
Memory
Writeback
(b)

Figure 7.45 Single-cycle and pipelined datapaths

7.5.2 Pipelined Control

The pipelined processor takes the same control signals as the single-cycle processor and therefore uses the same control unit. The control unit examines the `opcode` and `funct` fields of the instruction in the Decode stage to produce the control signals, as was described in Section 7.3.2. These control signals must be pipelined along with the data so that they remain synchronized with the instruction.

The entire pipelined processor with control is shown in Figure 7.47. *RegWrite* must be pipelined into the Writeback stage before it feeds back to the register file, just as *WriteReg* was pipelined in Figure 7.46.

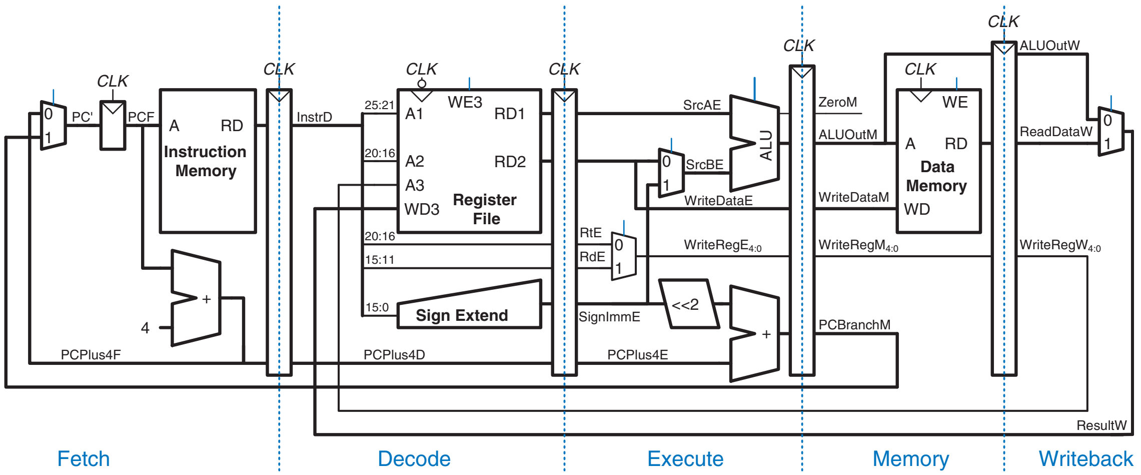

CLK
PC'
PCF
A
RD
Instruction Memory
4
+
PCPlus4F
InstrD
25:21
20:16
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
20:16
15:11
15:0
Sign Extend
PCPlus4D
SrcAE
SrcBE
0
1
ALU
WriteDataE
RtE
RdE
WriteRegE4:0
SignImmE
<<2
PCPlus4E
ZeroM
ALUOutM
WriteDataM
WriteRegM4:0
PCBranchM
WE
Data Memory
WD
ALUOutW
ReadDataW
WriteRegW4:0
ResultW
Fetch
Decode
Execute
Memory
Writeback

Figure 7.46 Corrected pipelined datapath

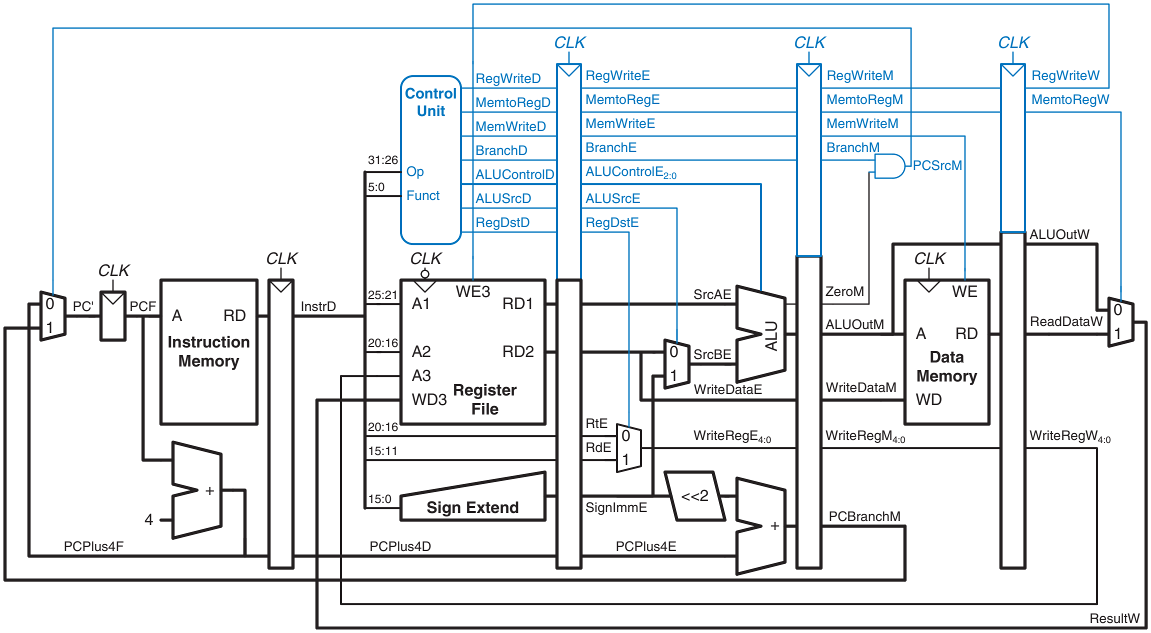

Control Unit
Op
Funct
31:26
5:0
RegWriteD
MemtoRegD
MemWriteD
BranchD
ALUControlD
ALUSrcD
RegDstD
RegWriteE
MemtoRegE
MemWriteE
BranchE
ALUControlE2:0
ALUSrcE
RegDstE
RegWriteM
MemtoRegM
MemWriteM
BranchM
PCSrcM
RegWriteW
MemtoRegW
CLK
PC'
PCF
A
RD
Instruction Memory
4
+
PCPlus4F
InstrD
25:21
20:16
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
20:16
15:11
15:0
Sign Extend
PCPlus4D
SrcAE
SrcBE
ALU
WriteDataE
RtE
RdE
WriteRegE4:0
SignImmE
<<2
PCPlus4E
ZeroM
ALUOutM
WriteDataM
WriteRegM4:0
PCBranchM
WE
Data Memory
WD
ALUOutW
ReadDataW
WriteRegW4:0
ResultW

Figure 7.47 Pipelined processor with control

7.5.3 Hazards

In a pipelined system, multiple instructions are handled concurrently. When one instruction is *dependent* on the results of another that has not yet completed, a *hazard* occurs.

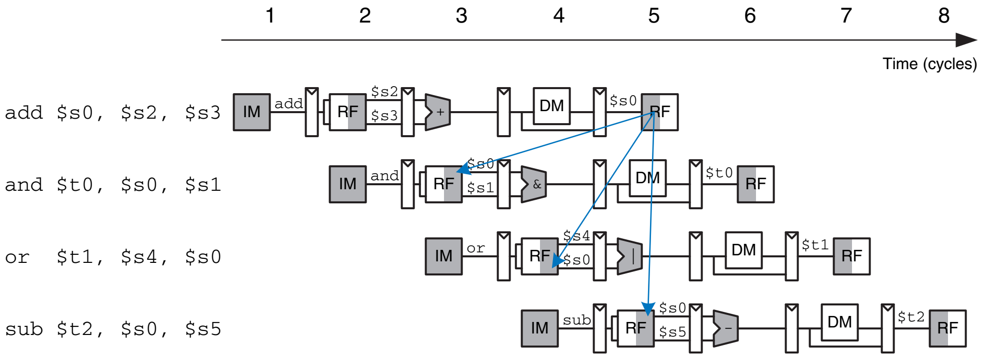


Figure 7.48 **Abstract pipeline diagram illustrating hazards**

The register file can be read and written in the same cycle. The write takes place during the first half of the cycle and the read takes place during the second half of the cycle, so a register can be written and read back in the same cycle without introducing a hazard.

Figure 7.48 illustrates hazards that occur when one instruction writes a register (`$s0`) and subsequent instructions read this register. This is called a *read after write* (*RAW*) hazard. The `add` instruction writes a result into `$s0` in the first half of cycle 5. However, the `and` instruction reads `$s0` on cycle 3, obtaining the wrong value. The `or` instruction reads `$s0` on cycle 4, again obtaining the wrong value. The `sub` instruction reads `$s0` in the second half of cycle 5, obtaining the correct value, which was written in the first half of cycle 5. Subsequent instructions also read the correct value of `$s0`. The diagram shows that hazards may occur in this pipeline when an instruction writes a register and either of the two subsequent instructions read that register. Without special treatment, the pipeline will compute the wrong result.

On closer inspection, however, observe that the sum from the `add` instruction is computed by the ALU in cycle 3 and is not strictly needed by the `and` instruction until the ALU uses it in cycle 4. In principle, we should be able to forward the result from one instruction to the next to resolve the RAW hazard without slowing down the pipeline. In other situations explored later in this section, we may have to stall the pipeline to give time for a result to be produced before the subsequent instruction uses the result. In any event, something must be done to solve hazards so that the program executes correctly despite the pipelining.

Hazards are classified as data hazards or control hazards. A *data hazard* occurs when an instruction tries to read a register that has not yet been written back by a previous instruction. A *control hazard* occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place. In the remainder of this section, we will enhance the pipelined processor with a hazard unit that detects hazards and handles them appropriately, so that the processor executes the program correctly.

Solving Data Hazards with Forwarding

Some data hazards can be solved by *forwarding* (also called *bypassing*) a result from the Memory or Writeback stage to a dependent instruction in the Execute stage. This requires adding multiplexers in front of the ALU to select the operand from either the register file or the Memory or Writeback stage. Figure 7.49 illustrates this principle. In cycle 4, `$s0` is forwarded from the Memory stage of the `add` instruction to the Execute stage of the dependent `and` instruction. In cycle 5, `$s0` is forwarded from the Writeback stage of the `add` instruction to the Execute stage of the dependent `or` instruction.

Forwarding is necessary when an instruction in the Execute stage has a source register matching the destination register of an instruction in the Memory or Writeback stage. Figure 7.50 modifies the pipelined processor to support forwarding. It adds a *hazard detection unit* and two forwarding multiplexers. The hazard detection unit receives the two source registers from the instruction in the Execute stage and the destination registers from the instructions in the Memory and Writeback stages. It also receives the *RegWrite* signals from the Memory and Writeback stages to know whether the destination register will actually be written (for example, the `sw` and `beq` instructions do not write results to the register file and hence do not need to have their results forwarded). Note that the *RegWrite* signals are *connected by name*. In other words, rather than cluttering up the diagram with long wires running from the control signals at the top to the hazard unit at the bottom, the connections are indicated by a short stub of wire labeled with the control signal name to which it is connected.

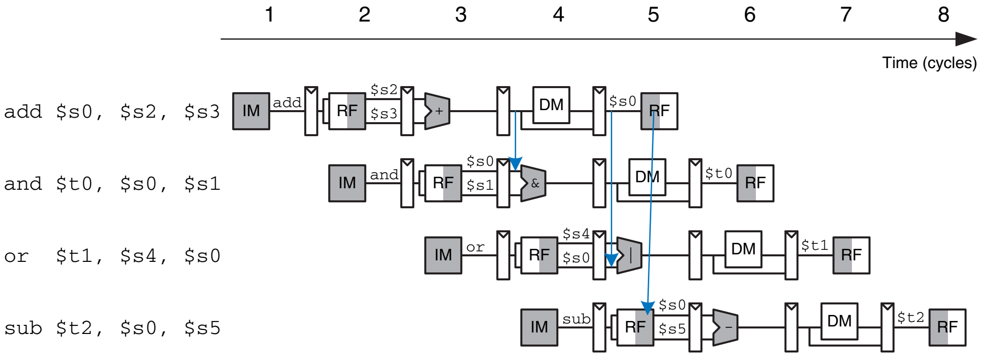


Figure 7.49 Abstract pipeline diagram illustrating forwarding

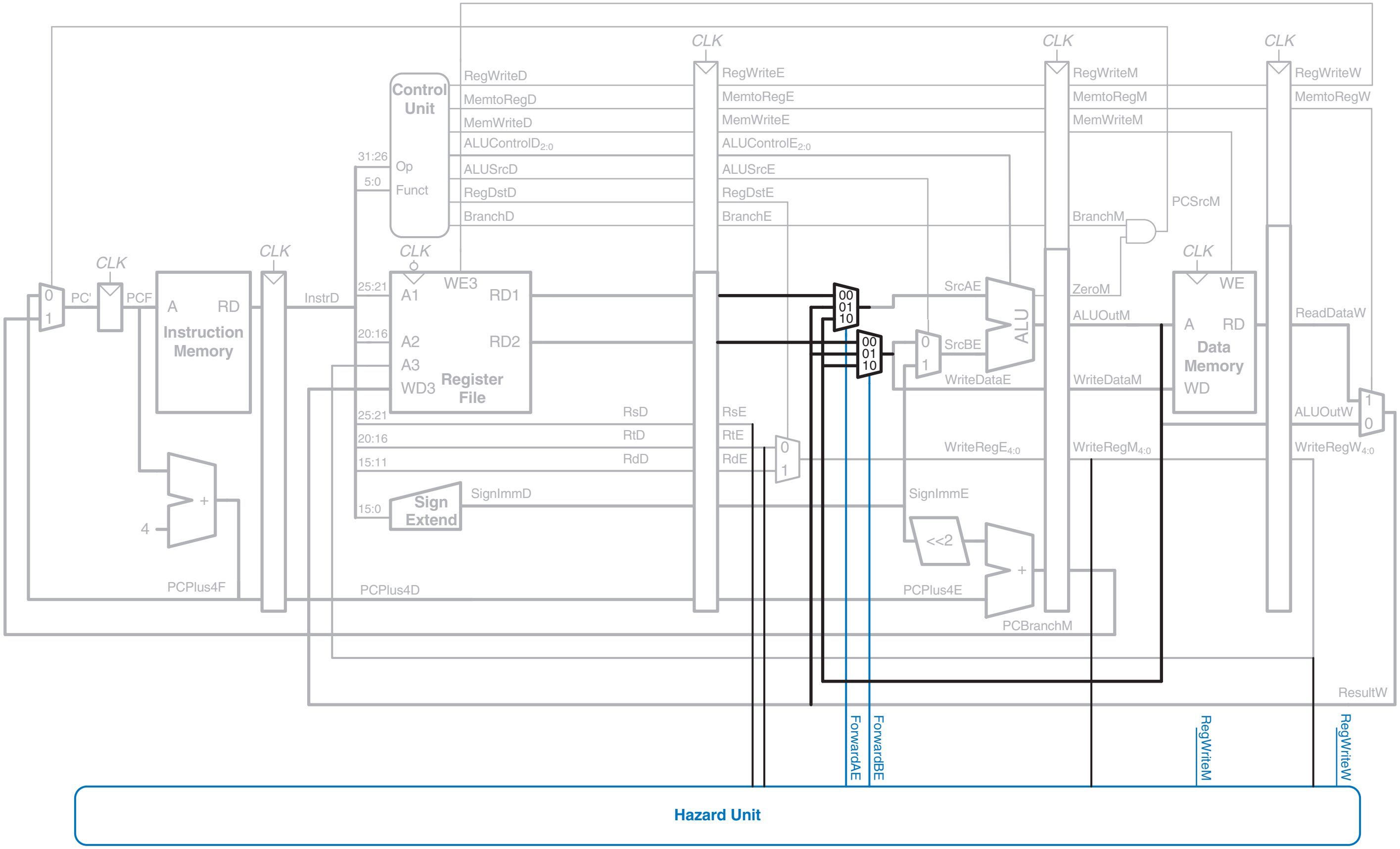

CLK
Control Unit
Op
Funct
31:26
5:0
RegWriteD
MemtoRegD
MemWriteD
$ALUControlD_{2:0}$
ALUSrcD
RegDstD
BranchD
RegWriteE
MemtoRegE
MemWriteE
$ALUControlE_{2:0}$
ALUSrcE
RegDstE
BranchE
RegWriteM
MemtoRegM
MemWriteM
BranchM
PCSrcM
RegWriteW
MemtoRegW
PC'
PCF
A
RD
Instruction Memory
InstrD
25:21
20:16
15:11
15:0
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
RsD
RtD
RdD
RsE
RtE
RdE
Sign Extend
SignImmD
SignImmE
SrcAE
SrcBE
WriteDataE
ALU
ZeroM
ALUOutM
WriteDataM
$WriteRegE_{4:0}$
$WriteRegM_{4:0}$
$WriteRegW_{4:0}$
WE
Data Memory
WD
ReadDataW
ALUOutW
ResultW
<<2
PCPlus4F
PCPlus4D
PCPlus4E
PCBranchM
4
+
00
01
10
0
1
ForwardAE
ForwardBE
RegWriteM
RegWriteW
Hazard Unit

Figure 7.50 Pipelined processor with forwarding to solve hazards

The hazard detection unit computes control signals for the forwarding multiplexers to choose operands from the register file or from the results in the Memory or Writeback stage. It should forward from a stage if that stage will write a destination register and the destination register matches the source register. However, `$0` is hardwired to 0 and should never be forwarded. If both the Memory and Writeback stages contain matching destination registers, the Memory stage should have priority, because it contains the more recently executed instruction. In summary, the function of the forwarding logic for *SrcA* is given below. The forwarding logic for *SrcB* (*ForwardBE*) is identical except that it checks `rt` rather than `rs`.

```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then
                                        ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then
                                        ForwardAE = 01
else                                    ForwardAE = 00
```

Solving Data Hazards with Stalls

Forwarding is sufficient to solve RAW data hazards when the result is computed in the Execute stage of an instruction, because its result can then be forwarded to the Execute stage of the next instruction. Unfortunately, the `lw` instruction does not finish reading data until the end of the Memory stage, so its result cannot be forwarded to the Execute stage of the next instruction. We say that the `lw` instruction has a *two-cycle latency*, because a dependent instruction cannot use its result until two cycles later. Figure 7.51 shows this problem. The `lw` instruction receives data from memory at the end of cycle 4. But the `and` instruction needs that data as a source operand at the beginning of cycle 4. There is no way to solve this hazard with forwarding.

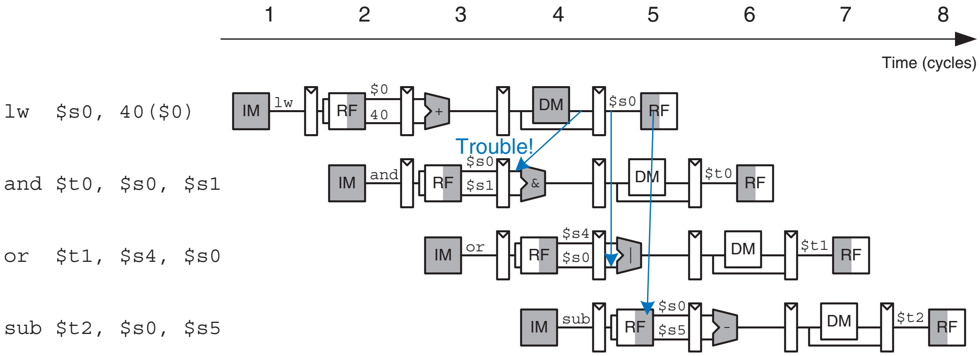


Figure 7.51 Abstract pipeline diagram illustrating trouble forwarding from `lw`

The alternative solution is to *stall* the pipeline, holding up operation until the data is available. Figure 7.52 shows stalling the dependent instruction (`and`) in the Decode stage. `and` enters the Decode stage in cycle 3 and stalls there through cycle 4. The subsequent instruction (`or`) must remain in the Fetch stage during both cycles as well, because the Decode stage is full.

In cycle 5, the result can be forwarded from the Writeback stage of `lw` to the Execute stage of `and`. In cycle 5, source `$s0` of the `or` instruction is read directly from the register file, with no need for forwarding.

Notice that the Execute stage is unused in cycle 4. Likewise, Memory is unused in Cycle 5 and Writeback is unused in cycle 6. This unused stage propagating through the pipeline is called a *bubble*, and it behaves like a `nop` instruction. The bubble is introduced by zeroing out the Execute stage control signals during a Decode stall so that the bubble performs no action and changes no architectural state.

In summary, stalling a stage is performed by disabling the pipeline register, so that the contents do not change. When a stage is stalled, all previous stages must also be stalled, so that no subsequent instructions are lost. The pipeline register directly after the stalled stage must be cleared to prevent bogus information from propagating forward. Stalls degrade performance, so they should only be used when necessary.

Figure 7.53 modifies the pipelined processor to add stalls for `lw` data dependencies. The hazard unit examines the instruction in the Execute stage. If it is `lw` and its destination register (`rtE`) matches either source operand of the instruction in the Decode stage (`rsD` or `rtD`), that instruction must be stalled in the Decode stage until the source operand is ready.

Stalls are supported by adding enable inputs (*EN*) to the Fetch and Decode pipeline registers and a synchronous reset/clear (*CLR*) input to the Execute pipeline register. When a `lw` stall occurs, *StallD* and *StallF*

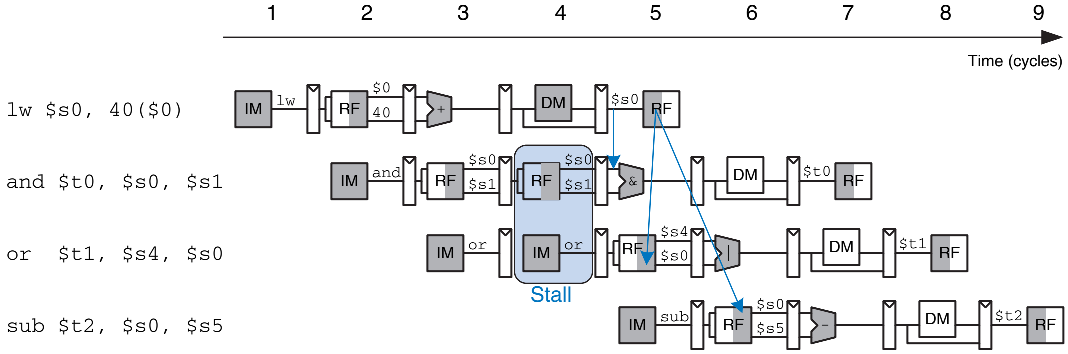


Figure 7.52 Abstract pipeline diagram illustrating stall to solve hazards

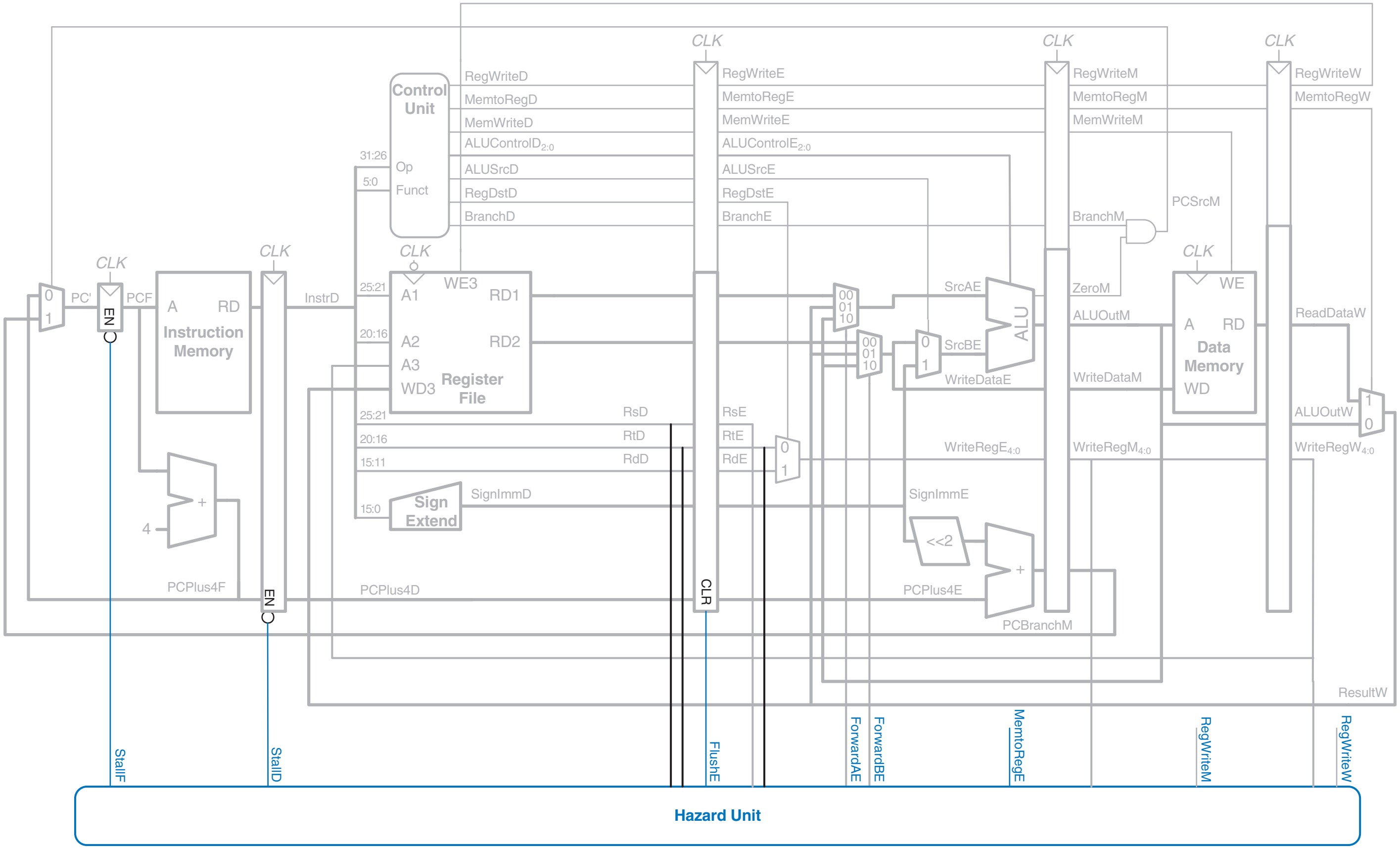

CLK
Control Unit
Op
Funct
31:26
5:0
RegWriteD
MemtoRegD
MemWriteD
ALUControlD$_{2:0}$
ALUSrcD
RegDstD
BranchD
RegWriteE
MemtoRegE
MemWriteE
ALUControlE$_{2:0}$
ALUSrcE
RegDstE
BranchE
RegWriteM
MemtoRegM
MemWriteM
BranchM
PCSrcM
RegWriteW
MemtoRegW
PC'
PCF
EN
A
RD
Instruction Memory
InstrD
25:21
20:16
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
RsD
RtD
RdD
15:11
15:0
Sign Extend
SignImmD
RsE
RtE
RdE
CLR
00
01
10
SrcAE
SrcBE
ALU
WriteDataE
WriteRegE$_{4:0}$
SignImmE
<<2
PCPlus4F
PCPlus4D
PCPlus4E
PCBranchM
4
ZeroM
ALUOutM
WriteDataM
WriteRegM$_{4:0}$
WE
Data Memory
WD
ReadDataW
ALUOutW
WriteRegW$_{4:0}$
ResultW
0
1
StallF
StallD
FlushE
ForwardAE
ForwardBE
MemtoRegE
RegWriteM
RegWriteW
Hazard Unit

Figure 7.53 Pipelined processor with stalls to solve `lw` data hazard

are asserted to force the Decode and Fetch stage pipeline registers to hold their old values. *FlushE* is also asserted to clear the contents of the Execute stage pipeline register, introducing a bubble.[4]

The *MemtoReg* signal is asserted for the `lw` instruction. Hence, the logic to compute the stalls and flushes is

```
lwstall = ((rsD == rtE) OR (rtD == rtE)) AND MemtoRegE
StallF  = StallD = FlushE = lwstall
```

Solving Control Hazards

The `beq` instruction presents a control hazard: the pipelined processor does not know what instruction to fetch next, because the branch decision has not been made by the time the next instruction is fetched.

One mechanism for dealing with the control hazard is to stall the pipeline until the branch decision is made (i.e., *PCSrc* is computed). Because the decision is made in the Memory stage, the pipeline would have to be stalled for three cycles at every branch. This would severely degrade the system performance.

An alternative is to predict whether the branch will be taken and begin executing instructions based on the prediction. Once the branch decision is available, the processor can throw out the instructions if the prediction was wrong. In particular, suppose that we predict that branches are not taken and simply continue executing the program in order. If the branch should have been taken, the three instructions following the branch must be *flushed* (discarded) by clearing the pipeline registers for those instructions. These wasted instruction cycles are called the *branch misprediction penalty*.

Figure 7.54 shows such a scheme, in which a branch from address 20 to address 64 is taken. The branch decision is not made until cycle 4, by which point the `and`, `or`, and `sub` instructions at addresses 24, 28, and 2C have already been fetched. These instructions must be flushed, and the `slt` instruction is fetched from address 64 in cycle 5. This is somewhat of an improvement, but flushing so many instructions when the branch is taken still degrades performance.

We could reduce the branch misprediction penalty if the branch decision could be made earlier. Making the decision simply requires comparing the values of two registers. Using a dedicated equality comparator is much faster than performing a subtraction and zero detection. If the comparator is fast enough, it could be moved back into the Decode stage, so that the operands are read from the register file and compared to determine the next PC by the end of the Decode stage.

---

[4] Strictly speaking, only the register designations (*RsE*, *RtE*, and *RdE*) and the control signals that might update memory or architectural state (*RegWrite*, *MemWrite*, and *Branch*) need to be cleared; as long as these signals are cleared, the bubble can contain random data that has no effect.

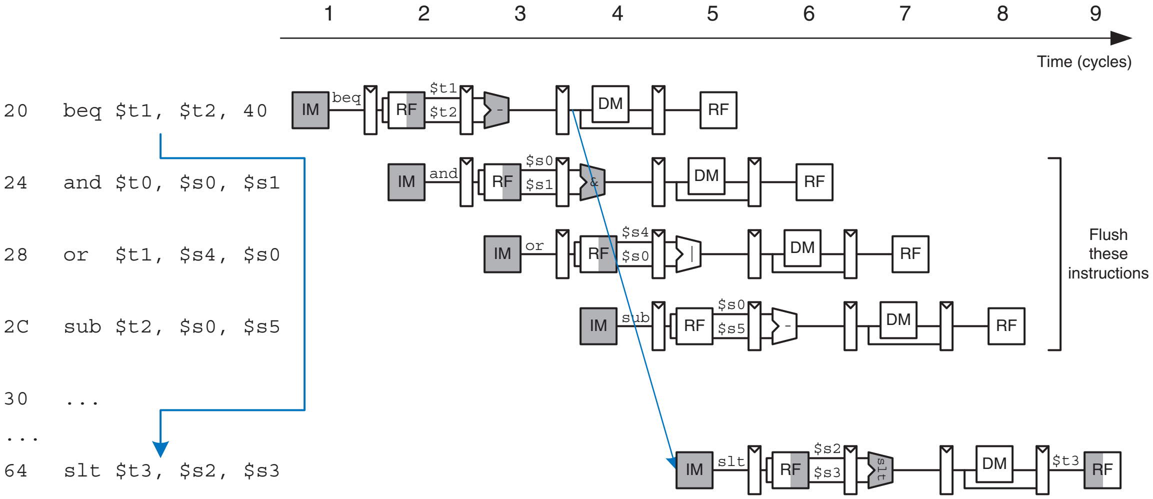

1 2 3 4 5 6 7 8 9
Time (cycles)
20 beq $t1, $t2, 40
24 and $t0, $s0, $s1
28 or $t1, $s4, $s0
2C sub $t2, $s0, $s5
30 ...
...
64 slt $t3, $s2, $s3
IM
beq
RF
$t1
$t2
-
DM
RF
and
$s0
$s1
&
or
$s4
$s0
|
sub
$s5
slt
$s2
$s3
$t3
Flush these instructions

Figure 7.54 Abstract pipeline diagram illustrating flushing when a branch is taken

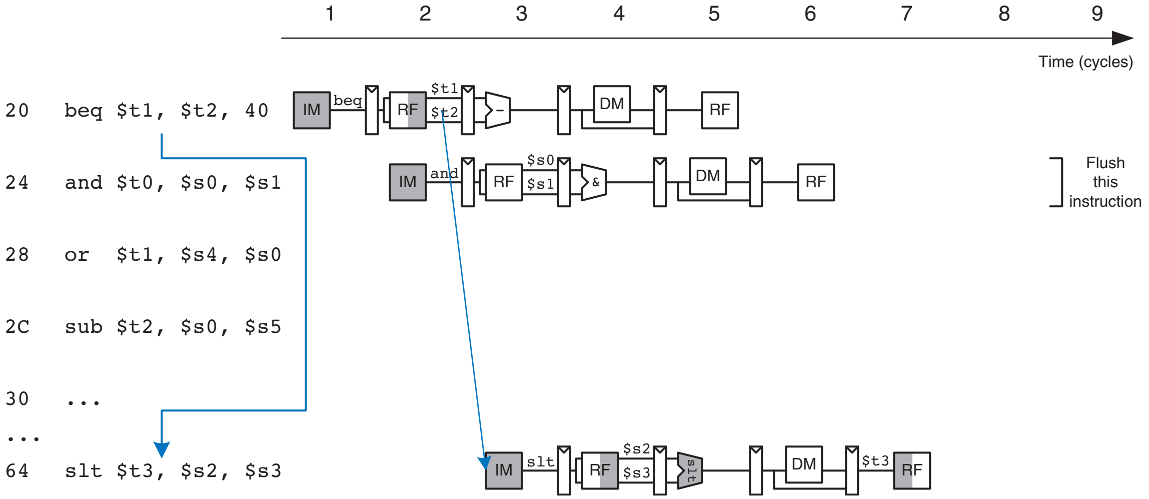

1 2 3 4 5 6 7 8 9
Time (cycles)
20 beq $t1, $t2, 40
24 and $t0, $s0, $s1
28 or $t1, $s4, $s0
2C sub $t2, $s0, $s5
30 ...
...
64 slt $t3, $s2, $s3
IM
beq
RF
$t1
$t2
-
DM
RF
and
$s0
$s1
&
slt
$s2
$s3
$t3
Flush this instruction

Figure 7.55 Abstract pipeline diagram illustrating earlier branch decision

Figure 7.55 shows the pipeline operation with the early branch decision being made in cycle 2. In cycle 3, the `and` instruction is flushed and the `slt` instruction is fetched. Now the branch misprediction penalty is reduced to only one instruction rather than three.

Figure 7.56 modifies the pipelined processor to move the branch decision earlier and handle control hazards. An equality comparator is added to the Decode stage and the *PCSrc* AND gate is moved earlier,

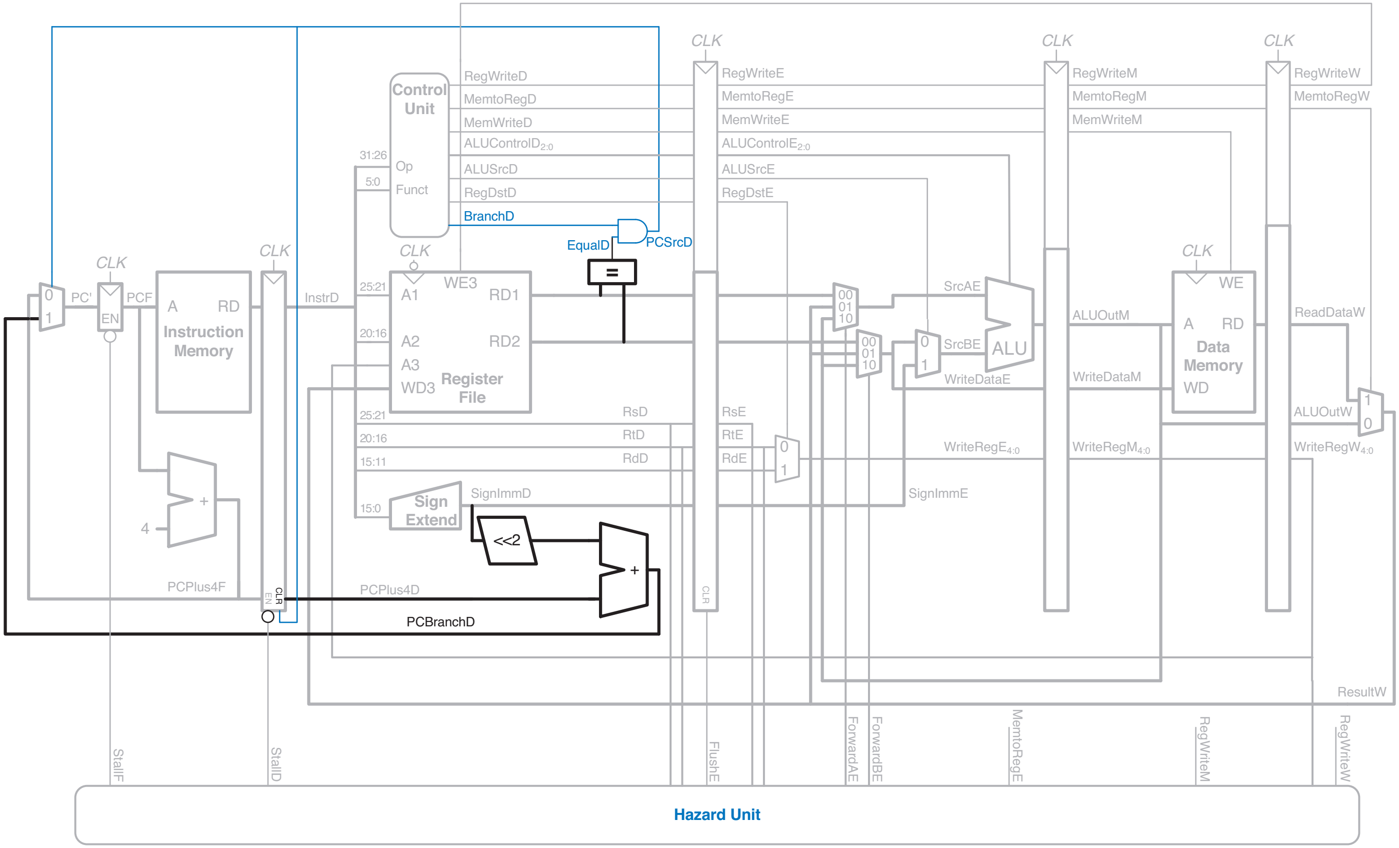

CLK
Control Unit
Op
Funct
31:26
5:0
RegWriteD
MemtoRegD
MemWriteD
ALUControlD$_{2:0}$
ALUSrcD
RegDstD
BranchD
EqualD
PCSrcD
RegWriteE
MemtoRegE
MemWriteE
ALUControlE$_{2:0}$
ALUSrcE
RegDstE
RegWriteM
MemtoRegM
MemWriteM
RegWriteW
MemtoRegW
PC'
PCF
EN
A
RD
Instruction Memory
InstrD
25:21
20:16
WE3
A1
A2
A3
WD3
RD1
RD2
Register File
=
SrcAE
SrcBE
ALU
00
01
10
0
1
WriteDataE
ALUOutM
WriteDataM
WE
Data Memory
WD
ReadDataW
ALUOutW
RsD
RtD
RdD
RsE
RtE
RdE
15:11
WriteRegE$_{4:0}$
WriteRegM$_{4:0}$
WriteRegW$_{4:0}$
15:0
Sign Extend
SignImmD
SignImmE
<<2
+
4
PCPlus4F
PCPlus4D
PCBranchD
CLR
ResultW
StallF
StallD
FlushE
ForwardAE
ForwardBE
MemtoRegE
RegWriteM
RegWriteW
Hazard Unit

Figure 7.56 Pipelined processor handling branch control hazard

so that *PCSrc* can be determined in the Decode stage rather than the Memory stage. The *PCBranch* adder must also be moved into the Decode stage so that the destination address can be computed in time. The synchronous clear input (*CLR*) connected to *PCSrcD* is added to the Decode stage pipeline register so that the incorrectly fetched instruction can be flushed when a branch is taken.

Unfortunately, the early branch decision hardware introduces a new RAW data hazard. Specifically, if one of the source operands for the branch was computed by a previous instruction and has not yet been written into the register file, the branch will read the wrong operand value from the register file. As before, we can solve the data hazard by forwarding the correct value if it is available or by stalling the pipeline until the data is ready.

Figure 7.57 shows the modifications to the pipelined processor needed to handle the Decode stage data dependency. If a result is in the Writeback stage, it will be written in the first half of the cycle and read during the second half, so no hazard exists. If the result of an ALU instruction is in the Memory stage, it can be forwarded to the equality comparator through two new multiplexers. If the result of an ALU instruction is in the Execute stage or the result of a `lw` instruction is in the Memory stage, the pipeline must be stalled at the Decode stage until the result is ready.

The function of the Decode stage forwarding logic is given below.

```
ForwardAD = (rsD != 0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD != 0) AND (rtD == WriteRegM) AND RegWriteM
```

The function of the stall detection logic for a branch is given below. The processor must make a branch decision in the Decode stage. If either of the sources of the branch depends on an ALU instruction in the Execute stage or on a `lw` instruction in the Memory stage, the processor must stall until the sources are ready.

```
branchstall =
   BranchD AND RegWriteE AND (WriteRegE == rsD OR WriteRegE == rtD)
                                   OR
   BranchD AND MemtoRegM AND (WriteRegM == rsD OR WriteRegM == rtD)
```

Now the processor might stall due to either a load or a branch hazard:

```
StallF = StallD = FlushE = lwstall OR branchstall
```

Hazard Summary

In summary, RAW data hazards occur when an instruction depends on the result of another instruction that has not yet been written into the register file. The data hazards can be resolved by forwarding if the result is computed soon enough; otherwise, they require stalling the pipeline until the result is available. Control hazards occur when the decision of what instruction to fetch has not been made by the time the next instruction

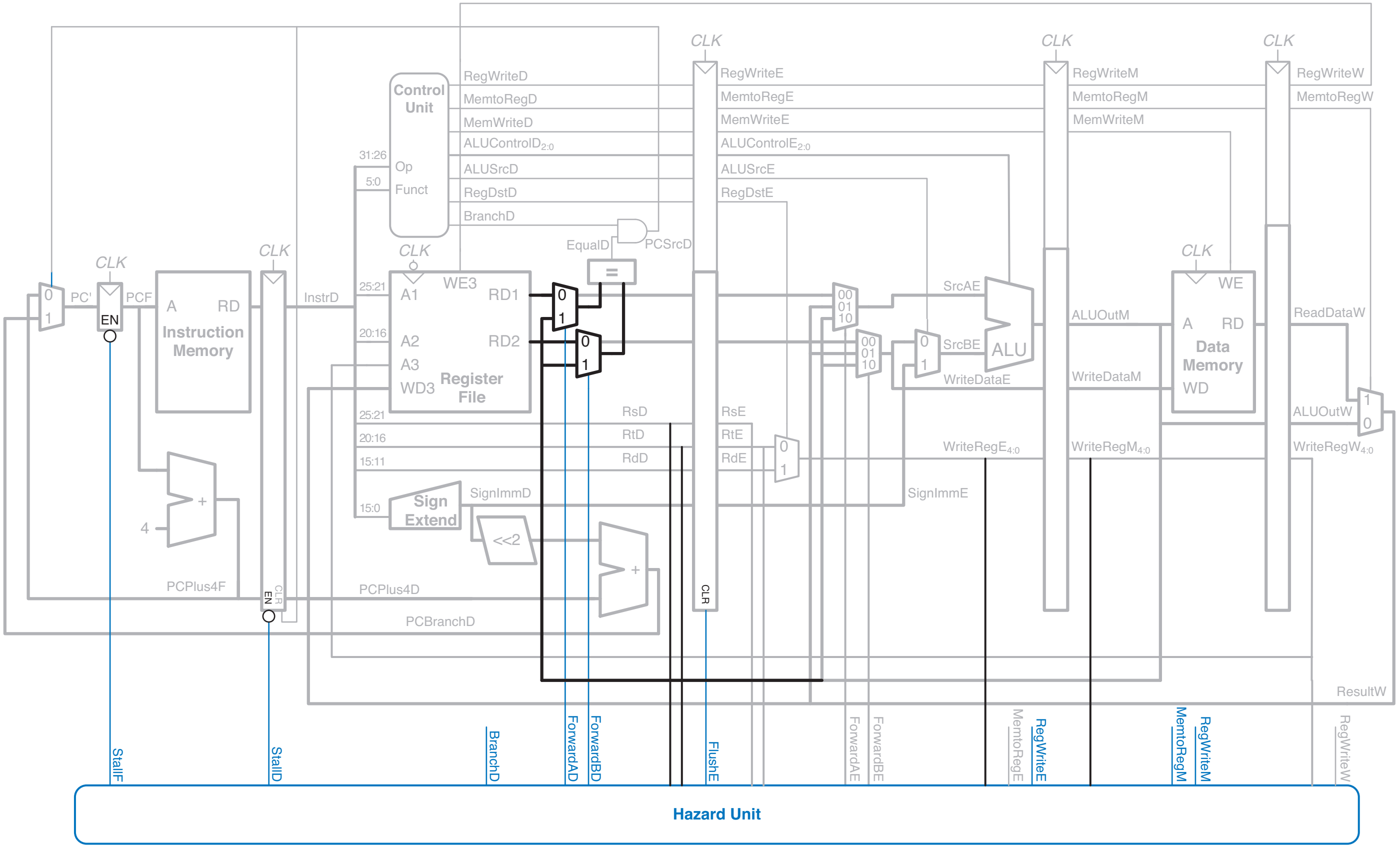

CLK
CLK
CLK
CLK
CLK
CLK
CLK
PC'
PCF
EN
A
RD
Instruction Memory
InstrD
Control Unit
Op
Funct
31:26
5:0
RegWriteD
MemtoRegD
MemWriteD
ALUControlD2:0
ALUSrcD
RegDstD
BranchD
EqualD
PCSrcD
=
25:21
20:16
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
25:21
20:16
15:11
15:0
RsD
RtD
RdD
Sign Extend
SignImmD
<<2
4
+
PCPlus4F
PCPlus4D
PCBranchD
CLR
EN
RegWriteE
MemtoRegE
MemWriteE
ALUControlE2:0
ALUSrcE
RegDstE
RsE
RtE
RdE
SrcAE
SrcBE
ALU
WriteDataE
WriteRegE4:0
SignImmE
00
01
10
0
1
RegWriteM
MemtoRegM
MemWriteM
ALUOutM
WriteDataM
WriteRegM4:0
WE
A
RD
Data Memory
WD
RegWriteW
MemtoRegW
ReadDataW
ALUOutW
WriteRegW4:0
ResultW
StallF
StallD
BranchD
ForwardAD
ForwardBD
FlushE
ForwardAE
ForwardBE
MemtoRegE
RegWriteE
MemtoRegM
RegWriteM
RegWriteW
Hazard Unit

Figure 7.57 Pipelined processor handling data dependencies for branch instructions

must be fetched. Control hazards are solved by predicting which instruction should be fetched and flushing the pipeline if the prediction is later determined to be wrong. Moving the decision as early as possible minimizes the number of instructions that are flushed on a misprediction. You may have observed by now that one of the challenges of designing a pipelined processor is to understand all the possible interactions between instructions and to discover all the hazards that may exist. Figure 7.58 shows the complete pipelined processor handling all of the hazards.

7.5.4 More Instructions

Supporting new instructions in the pipelined processor is much like supporting them in the single-cycle processor. However, new instructions may introduce hazards that must be detected and solved.

In particular, supporting `addi` and `j` instructions in the pipelined processor requires enhancing the controller, exactly as was described in Section 7.3.3, and adding a jump multiplexer to the datapath after the branch multiplexer. Like a branch, the jump takes place in the Decode stage, so the subsequent instruction in the Fetch stage must be flushed. Designing this flush logic is left as Exercise 7.35.

7.5.5 Performance Analysis

The pipelined processor ideally would have a CPI of 1, because a new instruction is issued every cycle. However, a stall or a flush wastes a cycle, so the CPI is slightly higher and depends on the specific program being executed.

Example 7.9 PIPELINED PROCESSOR CPI

The SPECINT2000 benchmark considered in Example 7.7 consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions. Assume that 40% of the loads are immediately followed by an instruction that uses the result, requiring a stall, and that one quarter of the branches are mispredicted, requiring a flush. Assume that jumps always flush the subsequent instruction. Ignore other hazards. Compute the average CPI of the pipelined processor.

Solution: The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of time that instruction is used. Loads take one clock cycle when there is no dependency and two cycles when the processor must stall for a dependency, so they have a CPI of $(0.6)(1) + (0.4)(2) = 1.4$. Branches take one clock cycle when they are predicted properly and two when they are not, so they have a CPI of $(0.75)(1) + (0.25)(2) = 1.25$. Jumps always have a CPI of 2. All other instructions have a CPI of 1. Hence, for this benchmark, Average CPI $= (0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1) = 1.15$.

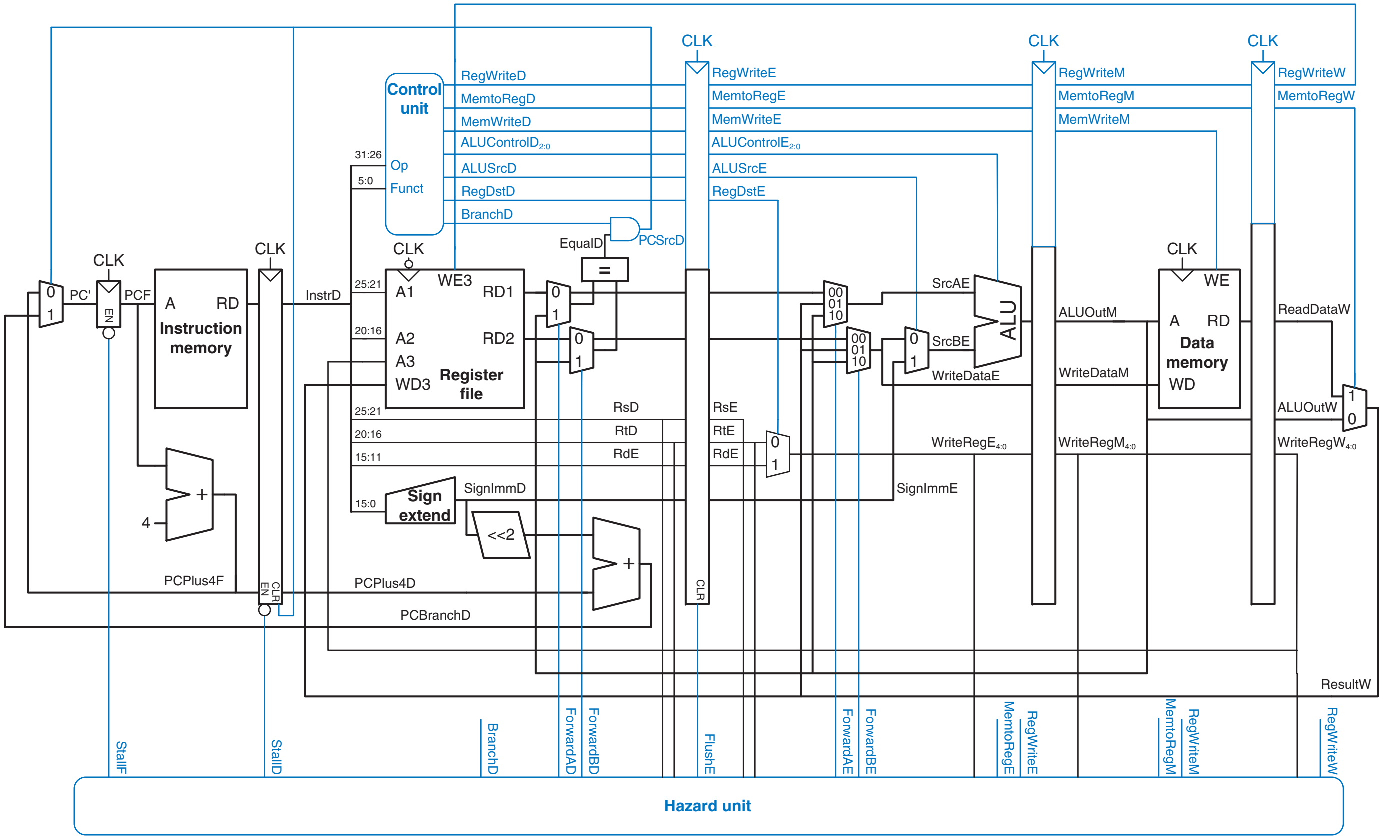

CLK
RegWriteD
MemtoRegD
MemWriteD
ALUControlD2:0
ALUSrcD
RegDstD
BranchD
Control unit
Op
Funct
31:26
5:0
EqualD
PCSrcD
CLK
WE3
A1
RD1
A2
RD2
A3
WD3
Register file
25:21
20:16
25:21
20:16
15:11
15:0
InstrD
CLK
PC'
PCF
A
RD
Instruction memory
EN
CLK
EN
CLR
4
+
PCPlus4F
PCPlus4D
PCBranchD
Sign extend
SignImmD
<<2
+
=
RsD
RtD
RdE
RsE
RtE
RdE
CLK
RegWriteE
MemtoRegE
MemWriteE
ALUControlE2:0
ALUSrcE
RegDstE
CLR
SrcAE
SrcBE
WriteDataE
ALU
00
01
10
WriteRegE4:0
SignImmE
CLK
RegWriteM
MemtoRegM
MemWriteM
ALUOutM
WriteDataM
WriteRegM4:0
CLK
WE
A
RD
Data memory
WD
CLK
RegWriteW
MemtoRegW
ReadDataW
ALUOutW
WriteRegW4:0
ResultW
StallF
StallD
BranchD
ForwardAD
ForwardBD
FlushE
ForwardAE
ForwardBE
MemtoRegE
RegWriteE
MemtoRegM
RegWriteM
RegWriteW
Hazard unit

Figure 7.58 Pipelined processor with full hazard handling

We can determine the cycle time by considering the critical path in each of the five pipeline stages shown in Figure 7.58. Recall that the register file is written in the first half of the Writeback cycle and read in the second half of the Decode cycle. Therefore, the cycle time of the Decode and Writeback stages is twice the time necessary to do the half-cycle of work.

$$T_c = max\left(\left.\begin{matrix} t_{pcq} + t_{\text{mem}} + t_{\text{setup}} \\ 2(t_{RF\text{read}} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + T_{\text{mux}} + t_{\text{setup}}) \\ t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{setup}} \\ t_{pcq} + t_{\text{memwrite}} + t_{\text{setup}} \\ 2(t_{pcq} + t_{\text{mux}} + t_{RF\text{write}}) \end{matrix}\right\} \begin{matrix} \text{Fetch} \\ \text{Decode} \\ \text{Execute} \\ \text{Memory} \\ \text{Writeback} \end{matrix}\right) \tag{7.5}$$

Example 7.10 PROCESSOR PERFORMANCE COMPARISON

Ben Bitdiddle needs to compare the pipelined processor performance to that of the single-cycle and multicycle processors considered in Example 7.8. Most of the logic delays were given in Table 7.6. The other element delays are 40 ps for an equality comparator, 15 ps for an AND gate, 100 ps for a register file write, and 220 ps for a memory write. Help Ben compare the execution time of 100 billion instructions from the SPECINT2000 benchmark for each processor.

Solution: According to Equation 7.5, the cycle time of the pipelined processor is $T_{c3} = \max[30+250+20,\ 2(150+25+40+15+25+20),\ 30+25+25+200+20,\ 30+220+20,\ 2(30+25+100)] = 550$ ps. According to Equation 7.1, the total execution time is $T_3 = (100 \times 10^9$ instructions)(1.15 cycles/instruction)$(550 \times 10^{-12}$ s/cycle) = 63.3 seconds. This compares to 92.5 seconds for the single-cycle processor and 133.9 seconds for the multicycle processor.

The pipelined processor is substantially faster than the others. However, its advantage over the single-cycle processor is nowhere near the fivefold speedup one might hope to get from a five-stage pipeline. The pipeline hazards introduce a small CPI penalty. More significantly, the sequencing overhead (clk-to-Q and setup times) of the registers applies to every pipeline stage, not just once to the overall datapath. Sequencing overhead limits the benefits one can hope to achieve from pipelining.

The careful reader might observe that the Decode stage is substantially slower than the others, because the register file read and branch comparison must both happen in half a cycle. Perhaps moving the branch comparison to the Decode stage was not such a good idea. If branches were resolved in the Execute stage instead, the CPI would increase slightly, because a mispredict would flush two instructions, but the cycle time would decrease substantially, giving an overall speedup.

The pipelined processor is similar in hardware requirements to the single-cycle processor, but it adds a substantial number of pipeline registers, along with multiplexers and control logic to resolve hazards.

7.6 HDL REPRESENTATION*

This section presents HDL code for the single-cycle MIPS processor supporting all of the instructions discussed in this chapter, including `addi` and `j`. The code illustrates good coding practices for a moderately complex system. HDL code for the multicycle processor and pipelined processor are left to Exercises 7.25 and 7.40.

In this section, the instruction and data memories are separated from the main processor and connected by address and data busses. This is more realistic, because most real processors have external memory. It also illustrates how the processor can communicate with the outside world.

The processor is composed of a datapath and a controller. The controller, in turn, is composed of the main decoder and the ALU decoder. Figure 7.59 shows a block diagram of the single-cycle MIPS processor interfaced to external memories.

The HDL code is partitioned into several sections. Section 7.6.1 provides HDL for the single-cycle processor datapath and controller. Section 7.6.2 presents the generic building blocks, such as registers and multiplexers, that are used by any microarchitecture. Section 7.6.3 introduces the

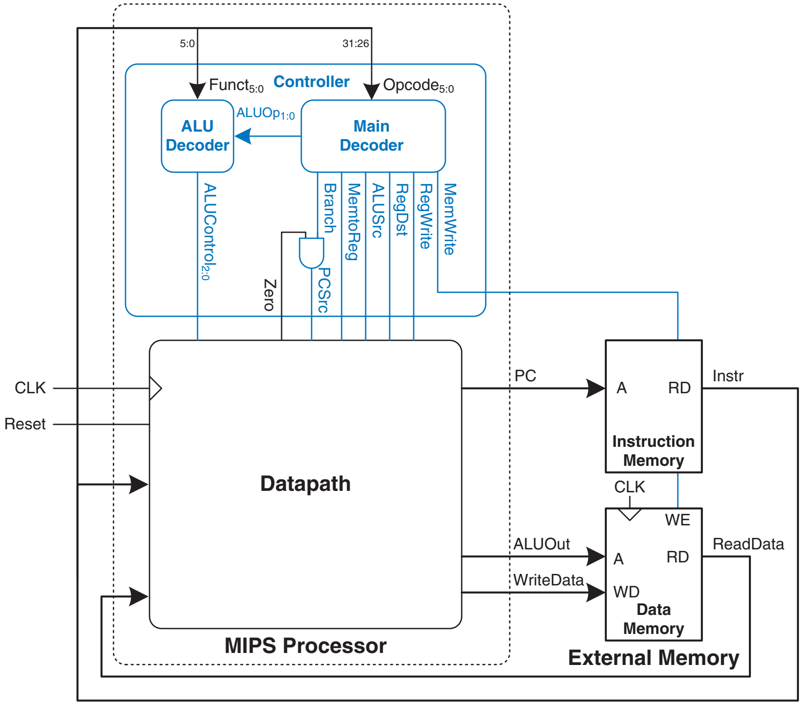

5:0
31:26
Funct$_{5:0}$
Controller
Opcode$_{5:0}$
ALUOp$_{1:0}$
ALU Decoder
Main Decoder
ALUControl$_{2:0}$
Branch
MemtoReg
ALUSrc
RegDst
RegWrite
MemWrite
Zero
PCSrc
CLK
Reset
Datapath
PC
A
RD
Instr
Instruction Memory
CLK
WE
ALUOut
A
RD
ReadData
WriteData
WD
Data Memory
MIPS Processor
External Memory

Figure 7.59 MIPS single-cycle processor interfaced to external memory

testbench and external memories. The HDL is available in electronic form on this book's website (see the preface).

7.6.1 Single-Cycle Processor

The main modules of the single-cycle MIPS processor module are given in the following HDL examples.

HDL Example 7.1 SINGLE-CYCLE MIPS PROCESSOR

SystemVerilog

```
module mips(input  logic        clk, reset,
            output logic [31:0] pc,
            input  logic [31:0] instr,
            output logic        memwrite,
            output logic [31:0] aluout, writedata,
            input  logic [31:0] readdata);

  logic        memtoreg, alusrc, regdst,
               regwrite, jump, pcsrc, zero;
  logic [2:0]  alucontrol;

  controller c(instr[31:26], instr[5:0], zero,
                memtoreg, memwrite, pcsrc,
                alusrc, regdst, regwrite, jump,
                alucontrol);
  datapath dp(clk, reset, memtoreg, pcsrc,
               alusrc, regdst, regwrite, jump,
               alucontrol,
               zero, pc, instr,
               aluout, writedata, readdata);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mips is -- single cycle MIPS processor
  port(clk, reset:        in  STD_LOGIC;
       pc:                out STD_LOGIC_VECTOR(31 downto 0);
       instr:             in  STD_LOGIC_VECTOR(31 downto 0);
       memwrite:          out STD_LOGIC;
       aluout, writedata: out STD_LOGIC_VECTOR(31 downto 0);
       readdata:          in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of mips is
  component controller
    port(op, funct:          in  STD_LOGIC_VECTOR(5 downto 0);
         zero:               in  STD_LOGIC;
         memtoreg, memwrite: out STD_LOGIC;
         pcsrc, alusrc:      out STD_LOGIC;
         regdst, regwrite:   out STD_LOGIC;
         jump:               out STD_LOGIC;
         alucontrol:         out STD_LOGIC_VECTOR(2 downto 0));
  end component;
  component datapath
    port(clk, reset:        in     STD_LOGIC;
         memtoreg, pcsrc:   in     STD_LOGIC;
         alusrc, regdst:    in     STD_LOGIC;
         regwrite, jump:    in     STD_LOGIC;
         alucontrol:        in     STD_LOGIC_VECTOR(2 downto 0);
         zero:              out    STD_LOGIC;
         pc:                buffer STD_LOGIC_VECTOR(31 downto 0);
         instr:             in     STD_LOGIC_VECTOR(31 downto 0);
         aluout, writedata: buffer STD_LOGIC_VECTOR(31 downto 0);
         readdata:          in     STD_LOGIC_VECTOR(31 downto 0));
  end component;
  signal memtoreg, alusrc, regdst, regwrite, jump, pcsrc: STD_LOGIC;
  signal zero: STD_LOGIC;
  signal alucontrol: STD_LOGIC_VECTOR(2 downto 0);
begin
  cont: controller port map(instr(31 downto 26),
                            instr(5 downto 0), zero, memtoreg,
                            memwrite, pcsrc, alusrc, regdst,
                            regwrite, jump, alucontrol);
  dp: datapath port map(clk, reset, memtoreg, pcsrc, alusrc,
                        regdst, regwrite, jump, alucontrol,
                        zero, pc, instr, aluout, writedata,
                        readdata);
end;
```

HDL Example 7.2 CONTROLLER

SystemVerilog

```
module controller(input  logic [5:0] op, funct,
                  input  logic       zero,
                  output logic       memtoreg, memwrite,
                  output logic       pcsrc, alusrc,
                  output logic       regdst, regwrite,
                  output logic       jump,
                  output logic [2:0] alucontrol);

  logic [1:0] aluop;
  logic       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump, aluop);
  aludec ad(funct, aluop, alucontrol);

  assign pcsrc = branch & zero;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity controller is -- single cycle control decoder
  port(op, funct:          in  STD_LOGIC_VECTOR(5 downto 0);
       zero:               in  STD_LOGIC;
       memtoreg, memwrite: out STD_LOGIC;
       pcsrc, alusrc:      out STD_LOGIC;
       regdst, regwrite:   out STD_LOGIC;
       jump:               out STD_LOGIC;
       alucontrol:         out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture struct of controller is
  component maindec
    port(op:                 in  STD_LOGIC_VECTOR(5 downto 0);
         memtoreg, memwrite: out STD_LOGIC;
         branch, alusrc:     out STD_LOGIC;
         regdst, regwrite:   out STD_LOGIC;
         jump:               out STD_LOGIC;
         aluop:              out  STD_LOGIC_VECTOR(1 downto 0));
  end component;
  component aludec
    port(funct:      in  STD_LOGIC_VECTOR(5 downto 0);
         aluop:      in  STD_LOGIC_VECTOR(1 downto 0);
         alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
  end component;
  signal aluop:  STD_LOGIC_VECTOR(1 downto 0);
  signal branch: STD_LOGIC;
begin
  md: maindec port map(op, memtoreg, memwrite, branch,
                       alusrc, regdst, regwrite, jump, aluop);
  ad: aludec port map(funct, aluop, alucontrol);

  pcsrc <= branch and zero;
end;
```

HDL Example 7.3 MAIN DECODER

SystemVerilog

```
module maindec(input  logic [5:0] op,
               output logic       memtoreg, memwrite,
               output logic       branch, alusrc,
               output logic       regdst, regwrite,
               output logic       jump,
               output logic [1:0] aluop);

  logic [8:0] controls;

  assign {regwrite, regdst, alusrc, branch, memwrite,
          memtoreg, jump, aluop} = controls;

  always_comb
    case(op)
      6'b000000: controls <= 9'b110000010; // RTYPE
      6'b100011: controls <= 9'b101001000; // LW
      6'b101011: controls <= 9'b001010000; // SW
      6'b000100: controls <= 9'b000100001; // BEQ
      6'b001000: controls <= 9'b101000000; // ADDI
      6'b000010: controls <= 9'b000000100; // J
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity maindec is -- main control decoder
  port(op:                 in  STD_LOGIC_VECTOR(5 downto 0);
       memtoreg, memwrite: out STD_LOGIC;
       branch, alusrc:     out STD_LOGIC;
       regdst, regwrite:   out STD_LOGIC;
       jump:               out STD_LOGIC;
       aluop:              out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture behave of maindec is
  signal controls: STD_LOGIC_VECTOR(8 downto 0);
begin
  process(all) begin
    case op is
      when "000000" => controls <= "110000010"; -- RTYPE
      when "100011" => controls <= "101001000"; -- LW
      when "101011" => controls <= "001010000"; -- SW
      when "000100" => controls <= "000100001"; -- BEQ
      when "001000" => controls <= "101000000"; -- ADDI
      when "000010" => controls <= "000000100"; -- J
      when others   => controls <= "---------"; -- illegal op
    end case;
  end process;

  (regwrite, regdst, alusrc, branch, memwrite,
   memtoreg, jump, aluop(1 downto 0)) <= controls;
end;
```

HDL Example 7.4 ALU DECODER

SystemVerilog

```
module aludec(input  logic [5:0] funct,
              input  logic [1:0] aluop,
              output logic [2:0] alucontrol);

  always_comb
    case(aluop)
      2'b00: alucontrol <= 3'b010; // add (for lw/sw/addi)
      2'b01: alucontrol <= 3'b110; // sub (for beq)
      default: case(funct)         // R-type instructions
          6'b100000: alucontrol <= 3'b010; // add
          6'b100010: alucontrol <= 3'b110; // sub
          6'b100100: alucontrol <= 3'b000; // and
          6'b100101: alucontrol <= 3'b001; // or
          6'b101010: alucontrol <= 3'b111; // slt
          default:   alucontrol <= 3'bxxx; // ???
        endcase
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity aludec is -- ALU control decoder
  port(funct:      in  STD_LOGIC_VECTOR(5 downto 0);
       aluop:      in  STD_LOGIC_VECTOR(1 downto 0);
       alucontrol: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture behave of aludec is
begin
  process(all) begin
    case aluop is
      when "00" => alucontrol <= "010"; -- add (for lw/sw/addi)
      when "01" => alucontrol <= "110"; -- sub (for beq)
      when others => case funct is      -- R-type instructions
        when "100000" => alucontrol <= "010"; -- add
        when "100010" => alucontrol <= "110"; -- sub
        when "100100" => alucontrol <= "000"; -- and
        when "100101" => alucontrol <= "001"; -- or
        when "101010" => alucontrol <= "111"; -- slt
        when others   => alucontrol <= "---"; -- ???
      end case;
    end case;
  end process;
end;
```

HDL Example 7.5 DATAPATH

SystemVerilog

```
module datapath(input  logic        clk, reset,
                input  logic        memtoreg, pcsrc,
                input  logic        alusrc, regdst,
                input  logic        regwrite, jump,
                input  logic [2:0]  alucontrol,
                output logic        zero,
                output logic [31:0] pc,
                input  logic [31:0] instr,
                output logic [31:0] aluout, writedata,
                input  logic [31:0] readdata);

  logic [4:0]  writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  logic [31:0] signimm, signimmsh;
  logic [31:0] srca, srcb;
  logic [31:0] result;

  // next PC   logic
  flopr #(32)  pcreg(clk, reset, pcnext, pc);
  adder        pcadd1(pc, 32'b100, pcplus4);
  sl2          immsh(signimm, signimmsh);
  adder        pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)   pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
  mux2 #(32)   pcmux(pcnextbr, {pcplus4[31:28],
                     instr[25:0], 2'b00}, jump, pcnext);

  // register file logic
  regfile     rf(clk, regwrite, instr[25:21], instr[20:16],
                 writereg, result, srca, writedata);
  mux2 #(5)   wrmux(instr[20:16], instr[15:11],
                    regdst, writereg);
  mux2 #(32)  resmux(aluout, readdata, memtoreg, result);
  signext     se(instr[15:0], signimm);

  // ALU logic
  mux2 #(32)   srcbmux(writedata, signimm, alusrc, srcb);
  alu          alu(srca, srcb, alucontrol, aluout, zero);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity datapath is -- MIPS datapath
  port(clk, reset:        in     STD_LOGIC;
       memtoreg, pcsrc:   in     STD_LOGIC;
       alusrc, regdst:    in     STD_LOGIC;
       regwrite, jump:    in     STD_LOGIC;
       alucontrol:        in     STD_LOGIC_VECTOR(2 downto 0);
       zero:              out    STD_LOGIC;
       pc:                buffer STD_LOGIC_VECTOR(31 downto 0);
       instr:             in     STD_LOGIC_VECTOR(31 downto 0);
       aluout, writedata: buffer STD_LOGIC_VECTOR(31 downto 0);
       readdata:          in     STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of datapath is
  component alu
    port(a, b:       in     STD_LOGIC_VECTOR(31 downto 0);
         alucontrol: in     STD_LOGIC_VECTOR(2 downto 0);
         result:     buffer STD_LOGIC_VECTOR(31 downto 0);
         zero:       out    STD_LOGIC);
  end component;
  component regfile
    port(clk:           in  STD_LOGIC;
         we3:           in  STD_LOGIC;
         ra1, ra2, wa3: in  STD_LOGIC_VECTOR(4 downto 0);
         wd3:           in  STD_LOGIC_VECTOR(31 downto 0);
         rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component adder
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:    out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component sl2
    port(a: in  STD_LOGIC_VECTOR(31 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component signext
    port(a: in  STD_LOGIC_VECTOR(15 downto 0);
         y: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flopr generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux2 generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
           s:    in  STD_LOGIC;
           y:    out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal  writereg:          STD_LOGIC_VECTOR(4 downto 0);
  signal  pcjump, pcnext,
          pcnextbr, pcplus4,
          pcbranch:          STD_LOGIC_VECTOR(31 downto 0);
  signal  signimm, signimmsh: STD_LOGIC_VECTOR(31 downto 0);
  signal  srca, srcb, result: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- next PC logic
  pcjump <= pcplus4(31 downto 28) & instr(25 downto 0) & "00";
  pcreg: flopr generic map(32) port map(clk, reset, pcnext, pc);
  pcadd1: adder port map(pc, X"00000004", pcplus4);
  immsh: sl2 port map(signimm, signimmsh);
  pcadd2: adder port map(pcplus4, signimmsh, pcbranch);
  pcbrmux: mux2 generic map(32) port map(pcplus4, pcbranch,
                                         pcsrc, pcnextbr);
  pcmux: mux2 generic map(32) port map(pcnextbr, pcjump, jump,
                                       pcnext);
  -- register file logic
  rf: regfile port map(clk, regwrite, instr(25 downto 21),
                       instr(20 downto 16), writereg, result, srca,
                       writedata);
  wrmux: mux2 generic map(5) port map(instr(20 downto 16),
                                      instr(15 downto 11),
                                      regdst, writereg);
  resmux: mux2 generic map(32) port map(aluout, readdata,
                                        memtoreg, result);
  se: signext port map(instr(15 downto 0), signimm);

  -- ALU logic
  srcbmux: mux2 generic map(32) port map(writedata, signimm,
                                         alusrc, srcb);
  mainalu: alu port map(srca, srcb, alucontrol, aluout, zero);
end;
```

7.6.2 Generic Building Blocks

This section contains generic building blocks that may be useful in any MIPS microarchitecture, including a register file, adder, left shift unit, sign-extension unit, resettable flip-flop, and multiplexer. The HDL for the ALU is left to Exercise 5.9.

HDL Example 7.6 REGISTER FILE

SystemVerilog

```
module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [4:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[31:0];

  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clk
  // register 0 hardwired to 0
  // note: for pipelined processor, write third port
  // on falling edge of clk

  always_ff @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity regfile is -- three-port register file
  port(clk:           in  STD_LOGIC;
       we3:           in  STD_LOGIC;
       ra1, ra2, wa3: in  STD_LOGIC_VECTOR(4 downto 0);
       wd3:           in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
  type ramtype is array (31 downto 0) of STD_LOGIC_VECTOR(31
                        downto 0);
  signal mem: ramtype;
begin
  -- three-ported register file
  -- read two ports combinationally
  -- write third port on rising edge of clk
  -- register 0 hardwired to 0
  -- note: for pipelined processor, write third port
  -- on falling edge of clk
  process(clk) begin
    if rising_edge(clk) then
      if we3 = '1' then mem(to_integer(wa3)) <= wd3;
      end if;
    end if;
  end process;
  process(all) begin
    if (to_integer(ra1) = 0) then rd1 <= X"00000000";
         -- register 0 holds 0
    else rd1 <= mem(to_integer(ra1));
    end if;
    if (to_integer(ra2) = 0) then rd2 <= X"00000000";
    else rd2 <= mem(to_integer(ra2));
    end if;
  end process;
end;
```

HDL Example 7.7 ADDER

SystemVerilog

```
module adder(input  logic [31:0] a, b,
             output logic [31:0] y);

  assign y = a + b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity adder is -- adder
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of adder is
begin
  y <= a + b;
end;
```

HDL Example 7.8 LEFT SHIFT (MULTIPLY BY 4)

SystemVerilog

```
module sl2(input  logic [31:0] a,
           output logic [31:0] y);
  // shift left by 2
  assign y = {a[29:0], 2'b00};
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sl2 is -- shift left by 2
  port(a: in  STD_LOGIC_VECTOR(31 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of sl2 is
begin
  y <= a(29 downto 0) & "00";
end;
```

HDL Example 7.9 SIGN EXTENSION

SystemVerilog

```
module signext(input  logic [15:0] a,
               output logic [31:0] y);
  assign y = {{16{a[15]}}, a};
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity signext is -- sign extender
  port(a: in  STD_LOGIC_VECTOR(15 downto 0);
       y: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of signext is
begin
  y <= X"ffff" & a when a(15) else X"0000" & a;
end;
```

HDL Example 7.10 RESETTABLE FLIP-FLOP

SystemVerilog

```
module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity flopr is -- flip-flop with synchronous reset
  generic (width: integer);
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:          out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then q <= (others => '0');
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;
```

HDL Example 7.11 2:1 MULTIPLEXER

SystemVerilog

```
module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is -- two-input multiplexer
  generic(width: integer := 8);
  port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux2 is
begin
  y <= d1 when s else d0;
end;
```

7.6.3 Testbench

The MIPS testbench loads a program into the memories. The program in Figure 7.60 exercises all of the instructions by performing a computation that should produce the correct answer only if all of the instructions are functioning properly. Specifically, the program will write the value 7 to address 84 if it runs correctly, and is unlikely to do so if the hardware is buggy. This is an example of *ad hoc* testing.

```
# mipstest.asm
# David_Harris@hmc.edu, Sarah_Harris@hmc.edu 31 March 2012
#
# Test the MIPS processor.
# add, sub, and, or, slt, addi, lw, sw, beq, j
# If successful, it should write the value 7 to address 84

#         Assembly            Description           Address   Machine
main:     addi $2, $0, 5      # initialize $2 = 5   0         20020005
          addi $3, $0, 12     # initialize $3 = 12  4         2003000c
          addi $7, $3, −9     # initialize $7 = 3   8         2067fff7
          or   $4, $7, $2     # $4 = (3 OR 5) = 7   c         00e22025
          and  $5, $3, $4     # $5 = (12 AND 7) = 4 10        00642824
          add  $5, $5, $4     # $5 = 4 + 7 = 11     14        00a42820
          beq  $5, $7, end    # shouldn't be taken  18        10a7000a
          slt  $4, $3, $4     # $4 = 12 < 7 = 0     1c        0064202a
          beq  $4, $0, around # should be taken     20        10800001
          addi $5, $0, 0      # shouldn’t happen    24        20050000
around:   slt  $4, $7, $2     # $4 = 3 < 5 = 1      28        00e2202a
          add  $7, $4, $5     # $7 = 1 + 11 = 12    2c        00853820
          sub  $7, $7, $2     # $7 = 12 − 5 = 7     30        00e23822
          sw   $7, 68($3)     # [80] = 7            34        ac670044
          lw   $2, 80($0)     # $2 = [80] = 7       38        8c020050
          j    end            # should be taken     3c        08000011
          addi $2, $0, 1      # shouldn't happen    40        20020001
end:      sw   $2, 84($0)     # write mem[84] = 7   44        ac020054
```

Figure 7.60 Assembly and machine code for MIPS test program

```
20020005
2003000c
2067fff7
00e22025
00642824
00a42820
10a7000a
0064202a
10800001
20050000
00e2202a
00853820
00e23822
ac670044
8c020050
08000011
20020001
ac020054
```

Figure 7.61 Contents of `memfile.dat`

The machine code is stored in a hexadecimal file called `memfile.dat` (see Figure 7.61), which is loaded by the testbench during simulation. The file consists of the machine code for the instructions, one instruction per line.

The testbench, top-level MIPS module, and external memory HDL code are given in the following examples. The memories in this example hold 64 words each.

HDL Example 7.12 MIPS TESTBENCH

SystemVerilog

```
module testbench();

  logic  clk;
  logic  reset;

  logic [31:0] writedata, dataadr;
  logic        memwrite;

  // instantiate device to be tested
  top dut (clk, reset, writedata, dataadr, memwrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @(negedge clk)
    begin
      if (memwrite) begin
        if (dataadr === 84 & writedata === 7) begin
          $display("Simulation succeeded");
          $stop;
        end else if (dataadr !== 80) begin
          $display("Simulation failed");
          $stop;
         end
       end
     end
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;

entity testbench is
end;

architecture test of testbench is
  component top
    port(clk, reset:         in  STD_LOGIC;
         writedata, dataadr: out STD_LOGIC_VECTOR(31 downto 0);
         memwrite:           out STD_LOGIC);
  end component;
  signal writedata, dataadr:   STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset, memwrite: STD_LOGIC;
begin

  -- instantiate device to be tested
  dut: top port map(clk, reset, writedata, dataadr, memwrite);

  -- Generate clock with 10 ns period
  process begin
    clk <= '1';
    wait for 5 ns;
    clk <= '0';
    wait for 5 ns;
  end process;

  -- Generate reset for first two clock cycles
  process begin
    reset <= '1';
    wait for 22 ns;
    reset <= '0';
    wait;
  end process;

  -- check that 7 gets written to address 84 at end of program
  process(clk) begin
    if (clk'event and clk = '0' and memwrite = '1') then
      if (to_integer(dataadr) = 84 and to_integer
          (writedata) = 7) then
        report "NO ERRORS: Simulation succeeded" severity failure;
      elsif (dataadr /= 80) then
        report "Simulation failed" severity failure;
      end if;
    end if;
  end process;
end;
```

HDL Example 7.13 MIPS TOP-LEVEL MODULE

SystemVerilog

```
module top(input  logic        clk, reset,
           output logic [31:0] writedata, dataadr,
           output logic        memwrite);

  logic [31:0] pc, instr, readdata;

  // instantiate processor and memories
  mips mips(clk, reset, pc, instr, memwrite, dataadr,
            writedata, readdata);
  imem imem(pc[7:2], instr);
  dmem dmem(clk, memwrite, dataadr, writedata, readdata);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;

entity top is -- top-level design for testing
  port(clk, reset:          in     STD_LOGIC;
       writedata, dataadr: buffer STD_LOGIC_VECTOR(31 downto 0);
       memwrite:            buffer STD_LOGIC);
end;

architecture test of top is
  component mips
    port(clk, reset:         in  STD_LOGIC;
         pc:                 out STD_LOGIC_VECTOR(31 downto 0);
         instr:              in  STD_LOGIC_VECTOR(31 downto 0);
         memwrite:           out STD_LOGIC;
         aluout, writedata: out STD_LOGIC_VECTOR(31 downto 0);
         readdata:           in  STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component imem
    port(a:  in  STD_LOGIC_VECTOR(5 downto 0);
         rd: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component dmem
    port(clk, we: in  STD_LOGIC;
         a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
         rd:      out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  signal pc, instr,
         readdata: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- instantiate processor and memories
  mips1: mips port map(clk, reset, pc, instr, memwrite,
                       dataadr, writedata, readdata);
  imem1: imem port map(pc(7 downto 2), instr);
  dmem1: dmem port map(clk, memwrite, dataadr, writedata, readdata);
end;
```

HDL Example 7.14 MIPS DATA MEMORY

SystemVerilog

```
module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity dmem is -- data memory
  port(clk, we: in  STD_LOGIC;
       a, wd:   in  STD_LOGIC_VECTOR (31 downto 0);
       rd:      out STD_LOGIC_VECTOR (31 downto 0));
end;

architecture behave of dmem is
begin
  process is
    type ramtype is array (63 downto 0) of
                          STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin
    -- read or write memory
    loop
      if rising_edge(clk) then
        if (we = '1') then mem (to_integer(a(7 downto 2))):= wd;
        end if;
      end if;
      rd <= mem (to_integer(a (7 downto 2)));
      wait on clk, a;
    end loop;

  end process;
end;
```

HDL Example 7.15 MIPS INSTRUCTION MEMORY

SystemVerilog

```
module imem(input  logic [5:0]  a,
            output logic [31:0] rd);

  logic [31:0] RAM[63:0];

  initial
    $readmemh("memfile.dat", RAM);

  assign rd = RAM[a]; // word aligned
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;

entity imem is -- instruction memory
  port(a:  in  STD_LOGIC_VECTOR(5 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of imem is
begin
  process is
    file mem_file: TEXT;
    variable L: line;
    variable ch: character;
    variable i, index, result: integer;
    type ramtype is array (63 downto 0) of
                         STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin
    -- initialize memory from file
    for i in 0 to 63 loop -- set all contents low
      mem(i) := (others => '0');
    end loop;
    index := 0;
    FILE_OPEN (mem_file, "C:/docs/DDCA2e/hdl/memfile.dat",
               READ_MODE);
    while not endfile(mem_file) loop
      readline(mem_file, L);
      result := 0;
      for i in 1 to 8 loop
        read (L, ch);
        if '0' <= ch and ch <= '9' then
            result := character'pos(ch) - character'pos('0');
        elsif 'a' <= ch and ch <= 'f' then
          result := character'pos(ch) - character'pos('a')+10;
        else report "Format error on line" & integer'
             image(index) severity error;
        end if;
        mem(index)(35-i*4 downto 32-i*4) :=
            to_std_logic_vector(result,4);
      end loop;
      index := index + 1;
    end loop;

    -- read memory
    loop
      rd <= mem(to_integer(a));
      wait on a;
    end loop;
  end process;
end;
```

7.7 EXCEPTIONS*

Section 6.7.2 introduced exceptions, which cause unplanned changes in the flow of a program. In this section, we enhance the multicycle processor to support two types of exceptions: undefined instructions and arithmetic overflow. Supporting exceptions in other microarchitectures follows similar principles.

As described in Section 6.7.2, when an exception takes place, the processor copies the PC to the EPC register and stores a code in the Cause register indicating the source of the exception. Exception causes include 0x28 for undefined instructions and 0x30 for overflow (see Table 6.7). The processor then jumps to the exception handler at memory address 0x80000180. The exception handler is code that responds to the exception. It is part of the operating system.

Also as discussed in Section 6.7.2, the exception registers are part of *Coprocessor 0*, a portion of the MIPS processor that is used for system functions. Coprocessor 0 defines up to 32 special-purpose registers, including Cause and EPC. The exception handler may use the `mfc0` (move from coprocessor 0) instruction to copy these special-purpose registers into a general-purpose register in the register file; the Cause register is Coprocessor 0 register 13, and EPC is register 14.

To handle exceptions, we must add EPC and Cause registers to the datapath and extend the *PCSrc* multiplexer to accept the exception handler address, as shown in Figure 7.62. The two new registers have write enables, *EPCWrite* and *CauseWrite*, to store the PC and exception cause when an exception takes place. The cause is generated by a multiplexer

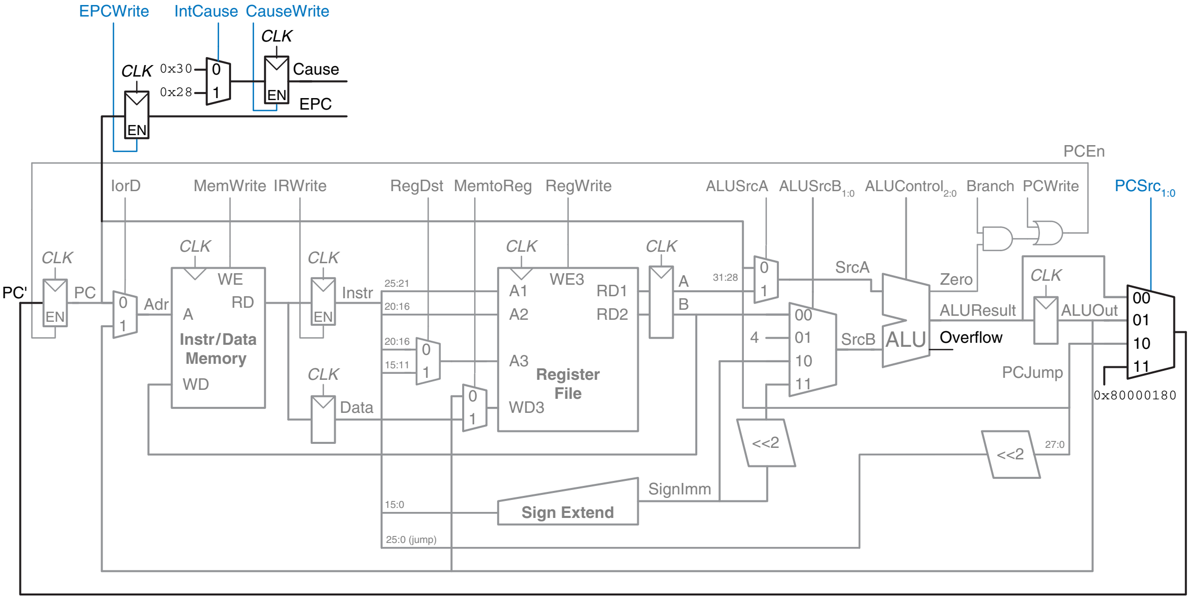

EPCWrite
IntCause
CauseWrite
CLK
0x30
0x28
0
1
Cause
EN
EPC
PCEn
IorD
MemWrite
IRWrite
RegDst
MemtoReg
RegWrite
ALUSrcA
ALUSrcB1:0
ALUControl2:0
Branch
PCWrite
PCSrc1:0
PC'
PC
Adr
A
WE
RD
Instr/Data Memory
WD
Instr
Data
25:21
20:16
15:11
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
31:28
SrcA
SrcB
4
00
01
10
11
ALU
Zero
ALUResult
Overflow
ALUOut
PCJump
0x80000180
<<2
27:0
SignImm
Sign Extend
15:0
25:0 (jump)

Figure 7.62 Datapath supporting overflow and undefined instruction exceptions

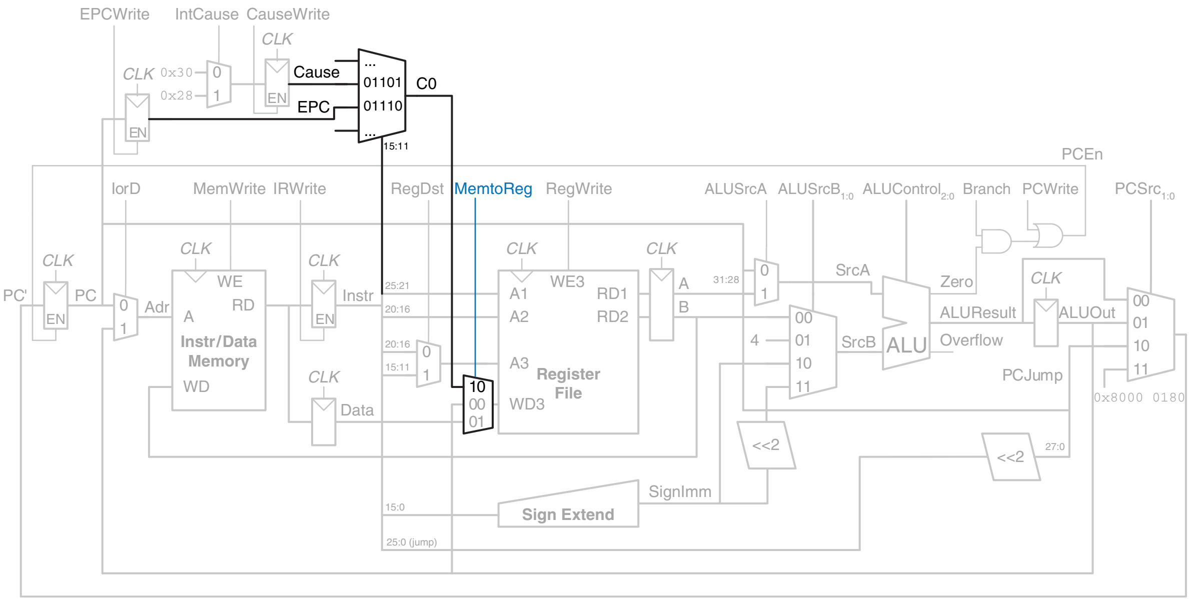

EPCWrite
IntCause
CauseWrite
CLK
CLK
0x30
0x28
0
1
EN
EN
Cause
EPC
...
01101
01110
...
C0
15:11
PCEn
IorD
MemWrite
IRWrite
RegDst
MemtoReg
RegWrite
ALUSrcA
ALUSrcB1:0
ALUControl2:0
Branch
PCWrite
PCSrc1:0
CLK
CLK
CLK
CLK
CLK
CLK
PC'
PC
EN
0
1
Adr
WE
RD
A
Instr/Data Memory
WD
Instr
EN
Data
25:21
20:16
20:16
15:11
0
1
10
00
01
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
A
B
31:28
0
1
SrcA
4
00
01
10
11
SrcB
ALU
Zero
ALUResult
Overflow
ALUOut
00
01
10
11
PCJump
0x8000 0180
<<2
<<2
27:0
SignImm
Sign Extend
15:0
25:0 (jump)

Figure 7.63 Datapath supporting `mfc0`

that selects the appropriate code for the exception. The ALU must also generate an overflow signal, as was discussed in Section 5.2.4.[5]

To support the `mfc0` instruction, we also add a way to select the Coprocessor 0 registers and write them to the register file, as shown in Figure 7.63. The `mfc0` instruction specifies the Coprocessor 0 register by $Instr_{15:11}$; in this diagram, only the Cause and EPC registers are supported. We add another input to the *MemtoReg* multiplexer to select the value from Coprocessor 0.

The modified controller is shown in Figure 7.64. The controller receives the overflow flag from the ALU. It generates three new control signals: one to write the EPC, a second to write the Cause register, and a third to select the Cause. It also includes two new states to support the two exceptions and another state to handle `mfc0`.

If the controller receives an undefined instruction (one that it does not know how to handle), it proceeds to S12, saves the PC in EPC, writes 0x28 to the Cause register, and jumps to the exception handler. Similarly, if the controller detects arithmetic overflow on an `add` or `sub` instruction, it proceeds to S13, saves the PC in EPC, writes 0x30 in the Cause register,

---

[5] Strictly speaking, the ALU should assert overflow only for `add` and `sub`, not for other ALU instructions.

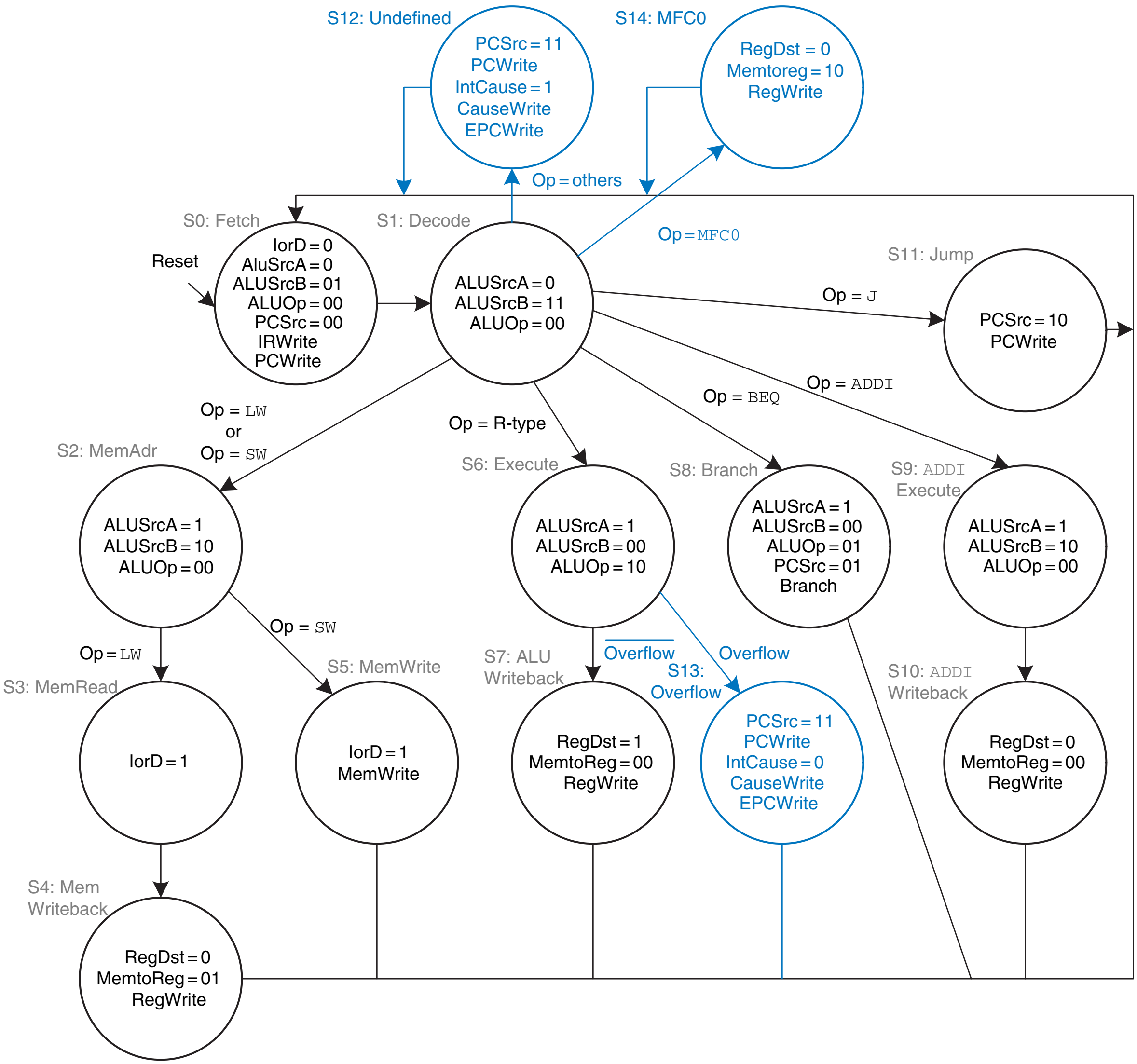

S12: Undefined
PCSrc = 11
PCWrite
IntCause = 1
CauseWrite
EPCWrite
S14: MFC0
RegDst = 0
Memtoreg = 10
RegWrite
Op = others
Op = MFC0
S0: Fetch
Reset
IorD = 0
AluSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCSrc = 00
IRWrite
PCWrite
S1: Decode
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
S11: Jump
Op = J
PCSrc = 10
PCWrite
Op = ADDI
Op = BEQ
Op = LW
or
Op = SW
Op = R-type
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
S6: Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
S8: Branch
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCSrc = 01
Branch
S9: ADDI
Execute
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Op = SW
Op = LW
S3: MemRead
IorD = 1
S5: MemWrite
IorD = 1
MemWrite
S7: ALU
Writeback
Overflow
Overflow
S13:
Overflow
RegDst = 1
MemtoReg = 00
RegWrite
PCSrc = 11
PCWrite
IntCause = 0
CauseWrite
EPCWrite
S10: ADDI
Writeback
RegDst = 0
MemtoReg = 00
RegWrite
S4: Mem
Writeback
RegDst = 0
MemtoReg = 01
RegWrite

Figure 7.64 **Controller supporting exceptions and** `mfc0`

and jumps to the exception handler. Note that, when an exception occurs, the instruction is discarded and the register file is not written. When a `mfc0` instruction is decoded, the processor goes to S14 and writes the appropriate Coprocessor 0 register to the main register file.

7.8 ADVANCED MICROARCHITECTURE*

High-performance microprocessors use a wide variety of techniques to run programs faster. Recall that the time required to run a program is proportional to the period of the clock and to the number of clock cycles per instruction (CPI). Thus, to increase performance we would like to speed up the clock and/or reduce the CPI. This section surveys some existing speedup techniques. The implementation details become quite complex, so we will focus on the concepts. Hennessy & Patterson's *Computer Architecture* text is a definitive reference if you want to fully understand the details.

Every 2 to 3 years, advances in CMOS manufacturing reduce transistor dimensions by 30% in each direction, doubling the number of transistors that can fit on a chip. A manufacturing process is characterized by its *feature size*, which indicates the smallest transistor that can be reliably built. Smaller transistors are faster and generally consume less power. Thus, even if the microarchitecture does not change, the clock frequency can increase because all the gates are faster. Moreover, smaller transistors enable placing more transistors on a chip. Microarchitects use the additional transistors to build more complicated processors or to put more processors on a chip. Unfortunately, power consumption increases with the number of transistors and the speed at which they operate (see Section 1.8). Power consumption is now an essential concern. Microprocessor designers have a challenging task juggling the trade-offs among speed, power, and cost for chips with billions of transistors in some of the most complex systems that humans have ever built.

7.8.1 Deep Pipelines

Aside from advances in manufacturing, the easiest way to speed up the clock is to chop the pipeline into more stages. Each stage contains less logic, so it can run faster. This chapter has considered a classic five-stage pipeline, but 10 to 20 stages are now commonly used.

The maximum number of pipeline stages is limited by pipeline hazards, sequencing overhead, and cost. Longer pipelines introduce more dependencies. Some of the dependencies can be solved by forwarding, but others require stalls, which increase the CPI. The pipeline registers between each stage have sequencing overhead from their setup time and clk-to-Q delay (as well as clock skew). This sequencing overhead makes adding more pipeline stages give diminishing returns. Finally, adding more stages increases the cost because of the extra pipeline registers and hardware required to handle hazards.

Example 7.11 DEEP PIPELINES

Consider building a pipelined processor by chopping up the single-cycle processor into N stages ($N \geq 5$). The single-cycle processor has a propagation delay of 875 ps through the combinational logic. The sequencing overhead of a register is 50 ps. Assume that the combinational delay can be arbitrarily divided into any number of stages and that pipeline hazard logic does not increase the delay. The five-stage pipeline in Example 7.9 has a CPI of 1.15. Assume that each additional stage increases the CPI by 0.1 because of branch mispredictions and other pipeline hazards. How many pipeline stages should be used to make the processor execute programs as fast as possible?

Solution: If the 875 ps combinational logic delay is divided into N stages and each stage also pays 50 ps of sequencing overhead for its pipeline register, the cycle time is $T_c = (875/N + 50)$ ps. The CPI is $1.15 + 0.1(N - 5)$. The time per instruction, or instruction time, is the product of the cycle time and the CPI. Figure 7.65 plots the cycle time and instruction time versus the number of stages. The instruction time has a minimum of 227 ps at $N = 11$ stages. This minimum is only slightly better than the 245 ps per instruction achieved with a six-stage pipeline.

In the late 1990s and early 2000s, microprocessors were marketed largely based on clock frequency ($1/T_c$). This pushed microprocessors to use very deep pipelines (20 to 31 stages on the Pentium 4) to maximize the clock frequency, even if the benefits for overall performance were questionable. Power is proportional to clock frequency and also increases with the number of pipeline registers, so now that power consumption is so important, pipeline depths are decreasing.

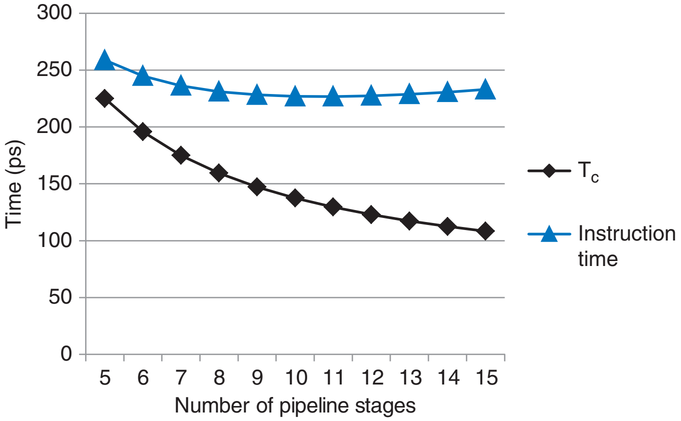


Figure 7.65 Cycle time and instruction time versus the number of pipeline stages

7.8.2 Branch Prediction

An ideal pipelined processor would have a CPI of 1. The branch misprediction penalty is a major reason for increased CPI. As pipelines get deeper, branches are resolved later in the pipeline. Thus, the branch misprediction penalty gets larger, because all the instructions issued after the mispredicted branch must be flushed. To address this problem, most pipelined processors use a *branch predictor* to guess whether the branch should be taken. Recall that our pipeline from Section 7.5.3 simply predicted that branches are never taken.

Some branches occur when a program reaches the end of a loop (e.g., a `for` or `while` statement) and branches back to repeat the loop. Loops tend to be executed many times, so these backward branches are usually taken. The simplest form of branch prediction checks the direction of the branch and predicts that backward branches should be taken. This is called *static branch prediction*, because it does not depend on the history of the program.

Forward branches are difficult to predict without knowing more about the specific program. Therefore, most processors use *dynamic branch predictors*, which use the history of program execution to guess whether a branch should be taken. Dynamic branch predictors maintain a table of the last several hundred (or thousand) branch instructions that the processor has executed. The table, sometimes called a *branch target buffer*, includes the destination of the branch and a history of whether the branch was taken.

To see the operation of dynamic branch predictors, consider the following loop code from Code Example 6.20. The loop repeats 10 times, and the `beq` out of the loop is taken only on the last time.

```
add  $s1,  $0,  $0     # sum = 0
addi $s0,  $0,  0      # i = 0
addi $t0,  $0,  10     # $t0 = 10

for:
   beq    $s0,  $t0,  done     # if i == 10, branch to done
   add    $s1,  $s1,  $s0      # sum = sum + i
   addi   $s0,  $s0,  1        # increment i
   j      for
done:
```

A *one-bit dynamic branch predictor* remembers whether the branch was taken the last time and predicts that it will do the same thing the next time. While the loop is repeating, it remembers that the `beq` was not taken last time and predicts that it should not be taken next time. This is a correct prediction until the last branch of the loop, when the branch does get taken. Unfortunately, if the loop is run again, the branch predictor remembers that the last branch was taken. Therefore, it incorrectly

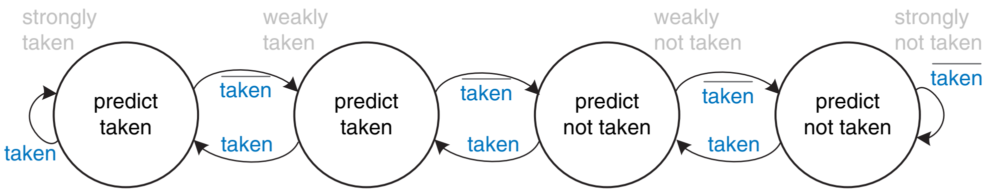


Figure 7.66 **2-bit branch predictor state transition diagram**

predicts that the branch should be taken when the loop is first run again. In summary, a 1-bit branch predictor mispredicts the first and last branches of a loop.

A 2-bit dynamic branch predictor solves this problem by having four states: *strongly taken*, *weakly taken*, *weakly not taken*, and *strongly not taken*, as shown in Figure 7.66. When the loop is repeating, it enters the "strongly not taken" state and predicts that the branch should not be taken next time. This is correct until the last branch of the loop, which is taken and moves the predictor to the "weakly not taken" state. When the loop is first run again, the branch predictor correctly predicts that the branch should not be taken and reenters the "strongly not taken" state. In summary, a 2-bit branch predictor mispredicts only the last branch of a loop.

As one can imagine, branch predictors may be used to track even more history of the program to increase the accuracy of predictions. Good branch predictors achieve better than 90% accuracy on typical programs.

The branch predictor operates in the Fetch stage of the pipeline so that it can determine which instruction to execute on the next cycle. When it predicts that the branch should be taken, the processor fetches the next instruction from the branch destination stored in the branch target buffer. By keeping track of both branch and jump destinations in the branch target buffer, the processor can also avoid flushing the pipeline during jump instructions.

7.8.3 Superscalar Processor

A *superscalar processor* contains multiple copies of the datapath hardware to execute multiple instructions simultaneously. Figure 7.67 shows a block diagram of a two-way superscalar processor that fetches and executes two instructions per cycle. The datapath fetches two instructions at a time from the instruction memory. It has a six-ported register file to read four source operands and write two results back in each cycle. It also contains two ALUs and a two-ported data memory to execute the two instructions at the same time.

A *scalar* processor acts on one piece of data at a time. A *vector* processor acts on several pieces of data with a single instruction. A *superscalar* processor issues several instructions at a time, each of which operates on one piece of data.

Our MIPS pipelined processor is a scalar processor. Vector processors were popular for supercomputers in the 1980s and 1990s because they efficiently handled the long vectors of data common in scientific computations. Modern high-performance microprocessors are superscalar, because issuing several independent instructions is more flexible than processing vectors.

However, modern processors also include hardware to handle short vectors of data that are common in multimedia and graphics applications. These are called *single instruction multiple data* (*SIMD*) units.

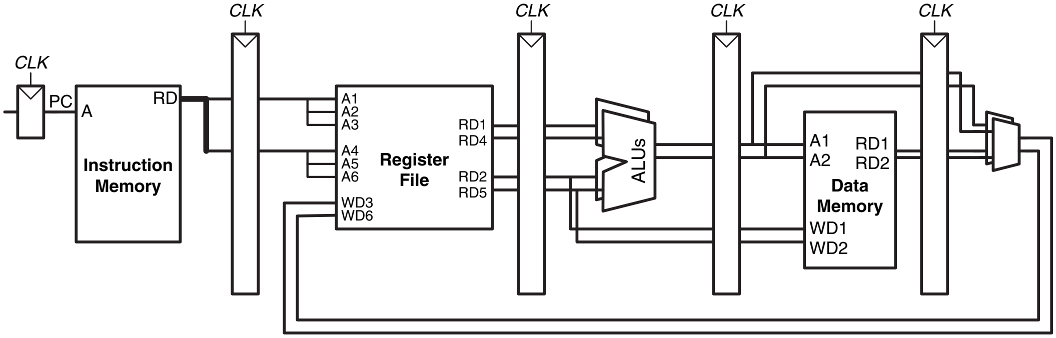

CLK
CLK
CLK
CLK
CLK
PC
A
RD
Instruction Memory
A1
A2
A3
A4
A5
A6
WD3
WD6
Register File
RD1
RD4
RD2
RD5
ALUs
A1
A2
RD1
RD2
Data Memory
WD1
WD2

Figure 7.67 Superscalar datapath

Figure 7.68 shows a pipeline diagram illustrating the two-way superscalar processor executing two instructions on each cycle. For this program, the processor has a CPI of 0.5. Designers commonly refer to the reciprocal of the CPI as the *instructions per cycle*, or *IPC*. This processor has an IPC of 2 on this program.

Executing many instructions simultaneously is difficult because of dependencies. For example, Figure 7.69 shows a pipeline diagram running a program with data dependencies. The dependencies in the code are shown in blue. The `add` instruction is dependent on `$t0`, which is produced by the `lw` instruction, so it cannot be issued at the same time as `lw`. Indeed, the `add` instruction stalls for yet another cycle so that `lw` can forward `$t0` to `add` in cycle 5. The other dependencies (between `sub` and `and`

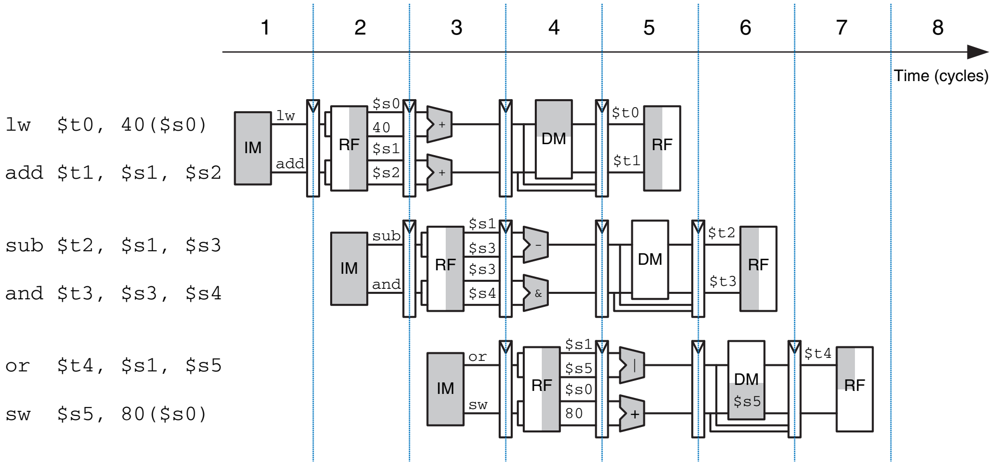

1
2
3
4
5
6
7
8
Time (cycles)
lw $t0, 40($s0)
add $t1, $s1, $s2
sub $t2, $s1, $s3
and $t3, $s3, $s4
or $t4, $s1, $s5
sw $s5, 80($s0)
IM
lw
add
RF
$s0
40
$s1
$s2
+
+
DM
$t0
$t1
RF
IM
sub
and
RF
$s1
$s3
$s3
$s4
-
&
DM
$t2
$t3
RF
IM
or
sw
RF
$s1
$s5
$s0
80
|
+
DM
$s5
$t4
RF

Figure 7.68 Abstract view of a superscalar pipeline in operation

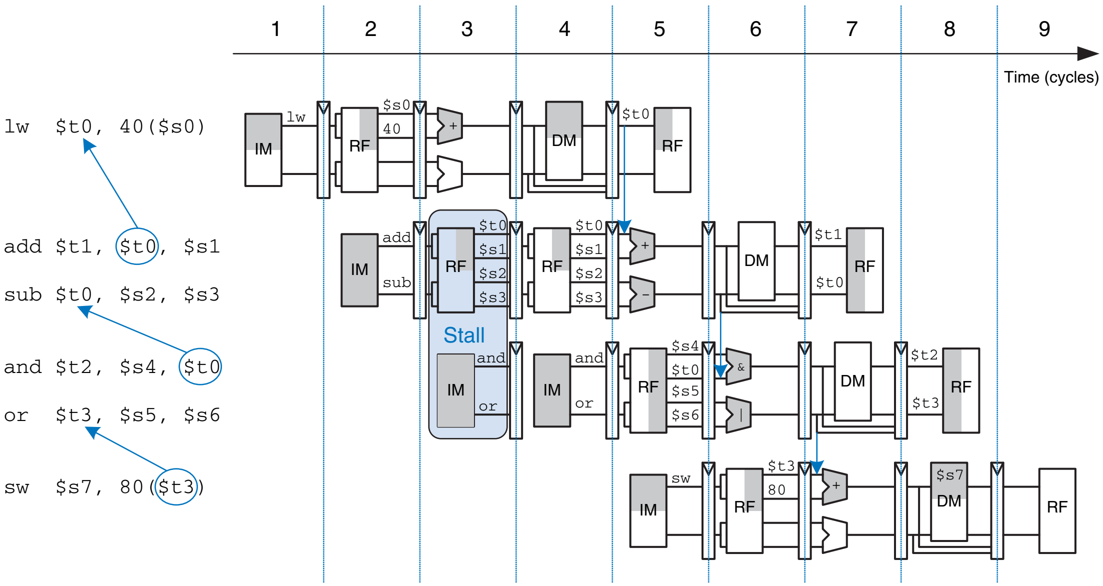

1
2
3
4
5
6
7
8
9
Time (cycles)
lw $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or $t3, $s5, $s6
sw $s7, 80($t3)
IM
lw
RF
$s0
40
+
DM
$t0
RF
IM
add
sub
RF
$t0
$s1
$s2
$s3
Stall
and
or
IM
RF
$t0
$s1
$s2
$s3
+
-
DM
$t1
$t0
RF
IM
and
or
RF
$s4
$t0
$s5
$s6
&
|
DM
$t2
$t3
RF
IM
sw
RF
$t3
80
+
$s7
DM
RF

Figure 7.69 Program with data dependencies

based on `$t0`, and between `or` and `sw` based on `$t3`) are handled by forwarding results produced in one cycle to be consumed in the next. This program, also given below, requires five cycles to issue six instructions, for an IPC of 1.17.

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

Recall that parallelism comes in temporal and spatial forms. Pipelining is a case of temporal parallelism. Multiple execution units is a case of spatial parallelism. Superscalar processors exploit both forms of parallelism to squeeze out performance far exceeding that of our single-cycle and multicycle processors.

Commercial processors may be three-, four-, or even six-way superscalar. They must handle control hazards such as branches as well as data hazards. Unfortunately, real programs have many dependencies, so wide superscalar processors rarely fully utilize all of the execution units. Moreover, the large number of execution units and complex forwarding networks consume vast amounts of circuitry and power.

7.8.4 Out-of-Order Processor

To cope with the problem of dependencies, an *out-of-order processor* looks ahead across many instructions to *issue*, or begin executing, independent instructions as rapidly as possible. The instructions can be issued in a different order than that written by the programmer, as long as dependencies are honored so that the program produces the intended result.

Consider running the same program from Figure 7.69 on a two-way superscalar out-of-order processor. The processor can issue up to two instructions per cycle from anywhere in the program, as long as dependencies are observed. Figure 7.70 shows the data dependencies and the operation of the processor. The classifications of dependencies as RAW and WAR will be discussed shortly. The constraints on issuing instructions are described below.

- **Cycle 1**
  - The `lw` instruction issues.
  - The `add`, `sub`, and `and` instructions are dependent on `lw` by way of `$t0`, so they cannot issue yet. However, the `or` instruction is independent, so it also issues.

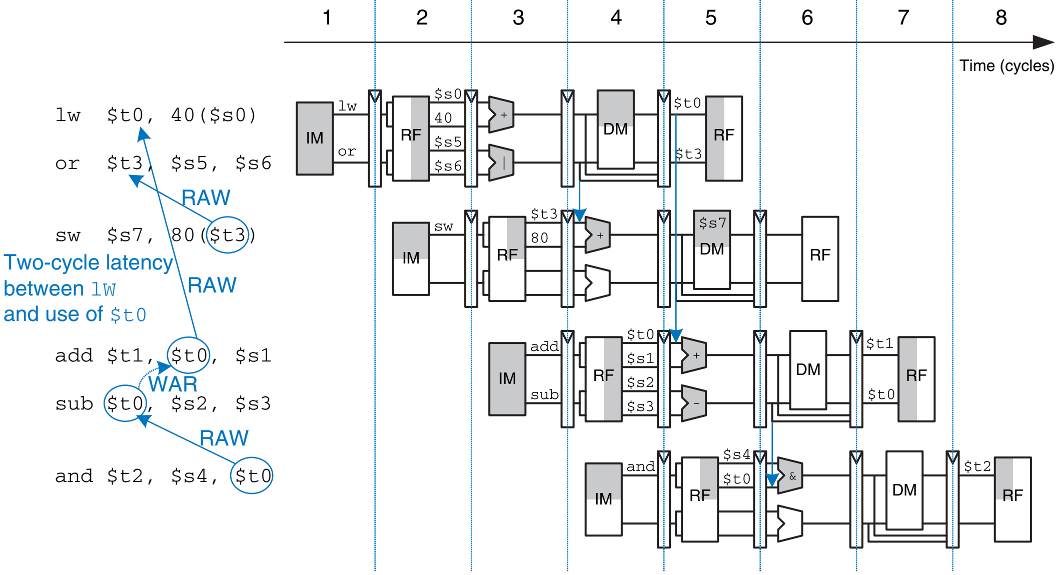

1
2
3
4
5
6
7
8
Time (cycles)
lw $t0, 40($s0)
or $t3, $s5, $s6
RAW
sw $s7, 80($t3)
Two-cycle latency between lw and use of $t0
RAW
add $t1, $t0, $s1
WAR
sub $t0, $s2, $s3
RAW
and $t2, $s4, $t0
IM
lw
or
RF
$s0
40
$s5
$s6
+
|
DM
$t0
$t3
RF
IM
sw
RF
$t3
80
+
$s7
DM
RF
IM
add
sub
RF
$t0
$s1
$s2
$s3
+
-
DM
$t1
$t0
RF
IM
and
RF
$s4
$t0
&
DM
$t2
RF

Figure 7.70 Out-of-order execution of a program with dependencies

- **Cycle 2**
  - Remember that there is a two-cycle latency between when a `lw` instruction issues and when a dependent instruction can use its result, so `add` cannot issue yet because of the `$t0` dependence. `sub` writes `$t0`, so it cannot issue before `add`, lest `add` receive the wrong value of `$t0`. `and` is dependent on `sub`.
  - Only the `sw` instruction issues.
- **Cycle 3**
  - On cycle 3, `$t0` is available, so `add` issues. `sub` issues simultaneously, because it will not write `$t0` until after `add` consumes `$t0`.
- **Cycle 4**
  - The `and` instruction issues. `$t0` is forwarded from `sub` to `and`.

The out-of-order processor issues the six instructions in four cycles, for an IPC of 1.5.

The dependence of `add` on `lw` by way of `$t0` is a read after write (RAW) hazard. `add` must not read `$t0` until after `lw` has written it. This is the type of dependency we are accustomed to handling in the pipelined processor. It inherently limits the speed at which the program can run, even if infinitely many execution units are available. Similarly, the dependence of `sw` on `or` by way of `$t3` and of `and` on `sub` by way of `$t0` are RAW dependencies.

The dependence between `sub` and `add` by way of `$t0` is called a *write after read* (*WAR*) hazard or an *antidependence*. `sub` must not write `$t0` before `add` reads `$t0`, so that `add` receives the correct value according to the original order of the program. WAR hazards could not occur in the simple MIPS pipeline, but they may happen in an out-of-order processor if the dependent instruction (in this case, `sub`) is moved too early.

A WAR hazard is not essential to the operation of the program. It is merely an artifact of the programmer's choice to use the same register for two unrelated instructions. If the `sub` instruction had written `$t4` instead of `$t0`, the dependency would disappear and `sub` could be issued before `add`. The MIPS architecture only has 32 registers, so sometimes the programmer is forced to reuse a register and introduce a hazard just because all the other registers are in use.

A third type of hazard, not shown in the program, is called *write after write* (*WAW*) or an *output dependence*. A WAW hazard occurs if an instruction attempts to write a register after a subsequent instruction has already written it. The hazard would result in the wrong value being written to the register. For example, in the following program, `add` and `sub` both write `$t0`. The final value in `$t0` should come from `sub` according to the order of the program. If an out-of-order processor attempted to execute `sub` first, a WAW hazard would occur.

```
add $t0, $s1, $s2
sub $t0, $s3, $s4
```

WAW hazards are not essential either; again, they are artifacts caused by the programmer's using the same register for two unrelated instructions. If the `sub` instruction were issued first, the program could eliminate the WAW hazard by discarding the result of the `add` instead of writing it to `$t0`. This is called *squashing* the `add`.[6]

Out-of-order processors use a table to keep track of instructions waiting to issue. The table, sometimes called a *scoreboard*, contains information about the dependencies. The size of the table determines how many instructions can be considered for issue. On each cycle, the processor examines the table and issues as many instructions as it can, limited by the dependencies and by the number of execution units (e.g., ALUs, memory ports) that are available.

The *instruction level parallelism* (*ILP*) is the number of instructions that can be executed simultaneously for a particular program and microarchitecture. Theoretical studies have shown that the ILP can be quite large for out-of-order microarchitectures with perfect branch predictors and enormous numbers of execution units. However, practical processors seldom achieve an ILP greater than 2 or 3, even with six-way superscalar datapaths with out-of-order execution.

7.8.5 Register Renaming

Out-of-order processors use a technique called *register renaming* to eliminate WAR hazards. Register renaming adds some nonarchitectural *renaming registers* to the processor. For example, a MIPS processor might add 20 renaming registers, called `$r0-$r19`. The programmer cannot use these registers directly, because they are not part of the architecture. However, the processor is free to use them to eliminate hazards.

For example, in the previous section, a WAR hazard occurred between the `sub` and `add` instructions based on reusing `$t0`. The out-of-order processor could *rename* `$t0` to `$r0` for the `sub` instruction. Then

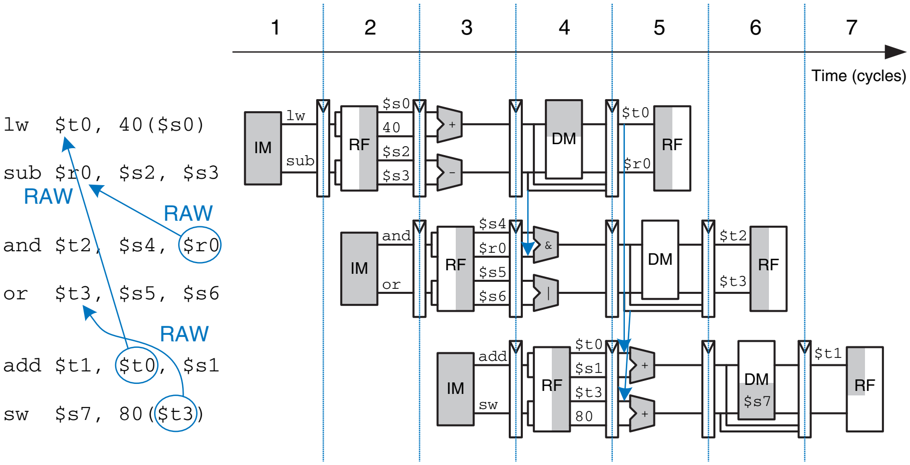
---

[6] You might wonder why the `add` needs to be issued at all. The reason is that out-of-order processors must guarantee that all of the same exceptions occur that would have occurred if the program had been executed in its original order. The `add` potentially may produce an overflow exception, so it must be issued to check for the exception, even though the result can be discarded.

`sub` could be executed sooner, because `$r0` has no dependency on the `add` instruction. The processor keeps a table of which registers were renamed so that it can consistently rename registers in subsequent dependent instructions. In this example, `$t0` must also be renamed to `$r0` in the `and` instruction, because it refers to the result of `sub`.

Figure 7.71 shows the same program from Figure 7.70 executing on an out-of-order processor with register renaming. `$t0` is renamed to `$r0` in `sub` and `and` to eliminate the WAR hazard. The constraints on issuing instructions are described below.

- Cycle 1
  - The `lw` instruction issues.
  - The `add` instruction is dependent on `lw` by way of `$t0`, so it cannot issue yet. However, the `sub` instruction is independent now that its destination has been renamed to `$r0`, so `sub` also issues.
- Cycle 2
  - Remember that there is a two-cycle latency between when a `lw` issues and when a dependent instruction can use its result, so `add` cannot issue yet because of the `$t0` dependence.
  - The `and` instruction is dependent on `sub`, so it can issue. `$r0` is forwarded from `sub` to `and`.
  - The `or` instruction is independent, so it also issues.



```
lw   $t0, 40($s0)

sub  $r0, $s2, $s3

and  $t2, $s4, $r0

or   $t3, $s5, $s6

add  $t1, $t0, $s1

sw   $s7, 80($t3)
```

Figure 7.71 Out-of-order execution of a program using register renaming

- Cycle 3
  - On cycle 3, `$t0` is available, so `add` issues. `$t3` is also available, so `sw` issues.

The out-of-order processor with register renaming issues the six instructions in three cycles, for an IPC of 2.

7.8.6 Single Instruction Multiple Data

The term *SIMD* (pronounced "sim-dee") stands for *single instruction multiple data*, in which a single instruction acts on multiple pieces of data in parallel. A common application of SIMD is to perform many short arithmetic operations at once, especially for graphics processing. This is also called *packed* arithmetic.

For example, a 32-bit microprocessor might pack four 8-bit data elements into one 32-bit word. Packed add and subtract instructions operate on all four data elements within the word in parallel. Figure 7.72 shows a packed 8-bit addition summing four pairs of 8-bit numbers to produce four results. The word could also be divided into two 16-bit elements. Performing packed arithmetic requires modifying the ALU to eliminate carries between the smaller data elements. For example, a carry out of $a_0 + b_0$ should not affect the result of $a_1 + b_1$.

Short data elements often appear in graphics processing. For example, a pixel in a digital photo may use 8 bits to store each of the red, green, and blue color components. Using an entire 32-bit word to process one of these components wastes the upper 24 bits. When the components from four adjacent pixels are packed into a 32-bit word, the processing can be performed four times faster.

SIMD instructions are even more helpful for 64-bit architectures, which can pack eight 8-bit elements, four 16-bit elements, or two 32-bit elements into a single 64-bit word. SIMD instructions are also used for floating-point computations; for example, four 32-bit single-precision floating-point values can be packed into a single 128-bit word.

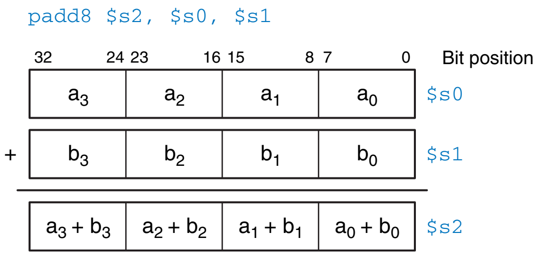


Figure 7.72 Packed arithmetic: four simultaneous 8-bit additions

7.8.7 Multithreading

Because the ILP of real programs tends to be fairly low, adding more execution units to a superscalar or out-of-order processor gives diminishing returns. Another problem, discussed in Chapter 8, is that memory is much slower than the processor. Most loads and stores access a smaller and faster memory, called a *cache*. However, when the instructions or data are not available in the cache, the processor may stall for 100 or more cycles while retrieving the information from the main memory. Multithreading is a technique that helps keep a processor with many execution units busy even if the ILP of a program is low or the program is stalled waiting for memory.

To explain multithreading, we need to define a few new terms. A program running on a computer is called a *process*. Computers can run multiple processes simultaneously; for example, you can play music on a PC while surfing the web and running a virus checker. Each process consists of one or more *threads* that also run simultaneously. For example, a word processor may have one thread handling the user typing, a second thread spell-checking the document while the user works, and a third thread printing the document. In this way, the user does not have to wait, for example, for a document to finish printing before being able to type again. The degree to which a process can be split into multiple threads that can run simultaneously defines its level of *thread level parallelism* (TLP).

In a conventional processor, the threads only give the illusion of running simultaneously. The threads actually take turns being executed on the processor under control of the OS. When one thread's turn ends, the OS saves its architectural state, loads the architectural state of the next thread, and starts executing that next thread. This procedure is called *context switching*. As long as the processor switches through all the threads fast enough, the user perceives all of the threads as running at the same time.

A multithreaded processor contains more than one copy of its architectural state, so that more than one thread can be active at a time. For example, if we extended a MIPS processor to have four program counters and 128 registers, four threads could be available at one time. If one thread stalls while waiting for data from main memory, the processor could context switch to another thread without any delay, because the program counter and registers are already available. Moreover, if one thread lacks sufficient parallelism to keep all the execution units busy, another thread could issue instructions to the idle units.

Multithreading does not improve the performance of an individual thread, because it does not increase the ILP. However, it does improve the overall throughput of the processor, because multiple threads can use processor resources that would have been idle when executing a single thread. Multithreading is also relatively inexpensive to implement, because it replicates only the PC and register file, not the execution units and memories.

7.8.8 Homogeneous Multiprocessors

A *multiprocessor* system consists of multiple processors and a method for communication between the processors. A common form of multiprocessing in computer systems is *homogeneous multiprocessing*, also called *symmetric multiprocessing* (SMP), in which two or more identical processors share a single main memory.

The multiple processors may be separate chips or multiple *cores* on the same chip. Modern processors have enormous numbers of transistors available. Using them to increase the pipeline depth or to add more execution units to a superscalar processor gives little performance benefit and is wasteful of power. Around the year 2005, computer architects made a major shift to building multiple copies of the processor on the same chip; these copies are called cores.

Multiprocessors can be used to run more threads simultaneously or to run a particular thread faster. Running more threads simultaneously is easy; the threads are simply divided up among the processors. Unfortunately typical PC users need to run only a small number of threads at any given time. Running a particular thread faster is much more challenging. The programmer must divide the existing thread into multiple threads to execute on each processor. This becomes tricky when the processors need to communicate with each other. One of the major challenges for computer designers and programmers is to effectively use large numbers of processor cores.

Other forms of multiprocessing include *heterogeneous multiprocessing* and *clusters*. Heterogeneous multiprocessors, also called *asymmetric multiprocessors*, use separate specialized microprocessors for separate tasks and are discussed next. In *clustered multiprocessing*, each processor has its own local memory system. Clustering can also refer to a group of PCs connected together on the network running software to jointly solve a large problem.

Scientists searching for signs of extraterrestrial intelligence use the world's largest clustered multiprocessors to analyze radio telescope data for patterns that might be signs of life in other solar systems. The cluster consists of personal computers owned by more than 3.8 million volunteers around the world.

When a computer in the cluster is idle, it fetches a piece of the data from a centralized server, analyzes the data, and sends the results back to the server. You can volunteer your computer's idle time for the cluster by visiting setiathome.berkeley.edu.

7.8.9 Heterogeneous Multiprocessors

The homogeneous multiprocessors described in Section 7.8.8 have a number of advantages. They are relatively simple to design because the processor can be designed once and then replicated multiple times to increase performance. Programming for and executing code on a homogeneous multiprocessor is also relatively straightforward because any program can run on any processor in the system and achieve approximately the same performance.

Unfortunately, continuing to add more and more cores is not guaranteed to provide continued performance improvement. As of 2012, consumer applications employed only 2–3 threads on average at any given time, and a typical consumer might be expected to have a couple of applications actually computing simultaneously. While this is enough to keep dual- and quad-core systems busy, unless programs start incorporating significantly more parallelism, continuing to add more cores beyond this point will provide diminishing benefits. As an added issue, because general purpose processors are designed to provide good average performance they are generally not the most power efficient option for performing a given operation. This energy inefficiency is especially important in highly power-constrained portable environments.

Heterogeneous multiprocessors aim to address these issues by incorporating different types of cores and/or specialized hardware in a single system. Each application uses those resources that provide the best performance, or power-performance ratio, for that application. Because transistors are fairly plentiful these days, the fact that not every application will make use of every piece of hardware is of minimal concern. Heterogeneous systems can take a number of forms. A heterogeneous system can incorporate cores with different microarchitectures that have different power, performance, and area tradeoffs. For example, a system could include both simple single-issue in-order cores and more complex superscalar out-of-order cores. Applications that can make efficient use of the higher performing, but more power hungry, out-of-order cores have that option available to them, while other applications that do not use the added computation effectively can use the more energy efficient in-order cores.

In such a system, the cores may all use the same ISA, which allows an application to run on any of the cores, or may employ different ISAs, which can allow further tailoring of a core to a given task. IBM's Cell Broadband Engine is an example of the latter. The Cell incorporates a single dual-issue in-order power processor element (PPE) with eight synergistic processor elements (SPEs). The SPEs use a new ISA, the *SPU ISA*, that is tailored to power efficiency for computationally intensive workloads. Although multiple programming paradigms are possible, the general idea is that the PPE handles most control and management decisions, such as dividing the workload among the SPEs, while the SPEs handle the majority of the computation. The heterogeneous nature of the Cell allows it to provide far higher computational performance for a given power and area than would be possible using traditional Power processors.

Other heterogeneous systems can include a mix of traditional cores and specialized hardware. Floating-point coprocessors are an early example of this. In early microprocessors, there was not space for floating-point hardware on the main chip. Users interested in floating-point performance could add a separate chip that provided dedicated floating-point support. Today's microprocessors include one or more floating-point units on chip and are now beginning to include other types of specialized hardware. AMD and Intel both have processors that incorporate a *graphics processing unit* (*GPU*) or FPGA and one or more traditional x86 cores on the same chip. AMD's Fusion line and Intel's Sandy Bridge are the first set of devices to incorporate a processor and GPU in the same die. Intel's E600 (Stellarton) series chips, released in early 2011, pair an Atom processor with an Altera FPGA, again on the same die. At a chip level, a cell phone contains both a conventional processor to handle user interaction, such as managing the phone book, processing websites and playing games, and a *digital signal processor* (*DSP*) with specialized instructions to decipher wireless data in real time. In power constrained environments, these types of integrated, specialized hardware provide better power-performance tradeoffs than performing the same operation on a standard core.

Heterogeneous systems are not without their drawbacks. They add complexity, both in terms of designing the different heterogeneous elements and in terms of the additional programming effort to decide when and how to make use of the varying resources. In the end, homogeneous and heterogeneous systems will both likely have their places. Homogeneous multiprocessors are good for situations, like large data centers, that have lots of thread level parallelism available. Heterogeneous systems are good for cases that have more varying workloads and limited parallelism.

Synergistic Processor Unit (SPU) ISA. The SPU ISA is designed to provide the same performance as general purpose processors at half the area and power for certain sets of workloads. Specifically, the SPU ISA targets graphics, stream processing, and other highly computational workloads, such as game physics and other system modeling. To achieve the desired performance, power, and area goals, the ISA incorporates a number of features, including 128-bit SIMD execution, software-managed memory, and a large register file. In software-managed memory the programmer is explicitly responsible for moving data into and out of local memory, as opposed to the caches of common processors, which bring data in automatically. If used correctly, this can both save power and improve performance because necessary data is brought in ahead of time and only when needed. The large register file helps avoid register starvation without the need for expensive register renaming.

7.9 REAL-WORLD PERSPECTIVE: x86 MICROARCHITECTURE*

Section 6.8 introduced the x86 architecture used in almost all PCs. This section tracks the evolution of x86 processors through progressively faster and more complicated microarchitectures. The same principles we have applied to the MIPS microarchitectures are used in x86.

Intel invented the first single-chip microprocessor, the 4-bit 4004, in 1971 as a flexible controller for a line of calculators. It contained 2300 transistors manufactured on a 12-mm^2 sliver of silicon in a process with a 10-μm feature size and operated at 750 kHz. A photograph of the chip taken under a microscope is shown in Figure 7.73.

In places, columns of four similar-looking structures are visible, as one would expect in a 4-bit microprocessor. Around the periphery are *bond wires*, which are used to connect the chip to its package and the circuit board.

The 4004 inspired the 8-bit 8008, then the 8080, which eventually evolved into the 16-bit 8086 in 1978 and the 80286 in 1982. In 1985, Intel introduced the 80386, which extended the 8086 architecture to 32 bits and defined the x86 architecture. Table 7.7 summarizes major

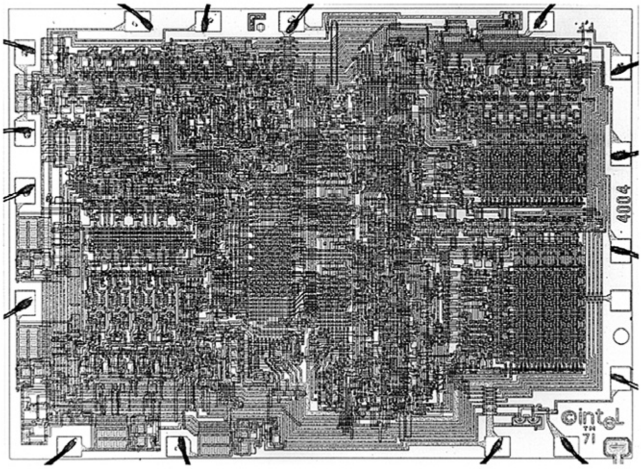


Figure 7.73 **4004 microprocessor chip**

Intel x86 microprocessors. In the 40 years since the 4004, transistor feature size has shrunk 160-fold, the number of transistors on a chip has increased by five orders of magnitude, and the operating frequency has increased by almost four orders of magnitude. No other field of engineering has made such astonishing progress in such a short time.

The 80386 is a multicycle processor. The major components are labeled on the chip photograph in Figure 7.74. The 32-bit datapath is

Table 7.7 **Evolution of Intel x86 microprocessors**

Processor	Year	Feature Size (μm)	Transistors	Frequency (MHz)	Microarchitecture
80386	1985	1.5–1.0	275k	16–25	multicycle
80486	1989	1.0–0.6	1.2M	25–100	pipelined
Pentium	1993	0.8–0.35	3.2–4.5M	60–300	superscalar
Pentium II	1997	0.35–0.25	7.5M	233–450	out of order
Pentium III	1999	0.25–0.18	9.5M–28M	450–1400	out of order
Pentium 4	2001	0.18–0.09	42–178M	1400–3730	out of order
Pentium M	2003	0.13–0.09	77–140M	900–2130	out of order
Core Duo	2005	0.065	152M	1500–2160	dual core
Core 2 Duo	2006	0.065–0.045	167–410M	1800–3300	dual core
Core i-series	2009	0.045–0.032	382–731$^+$M	2530–3460	multi-core

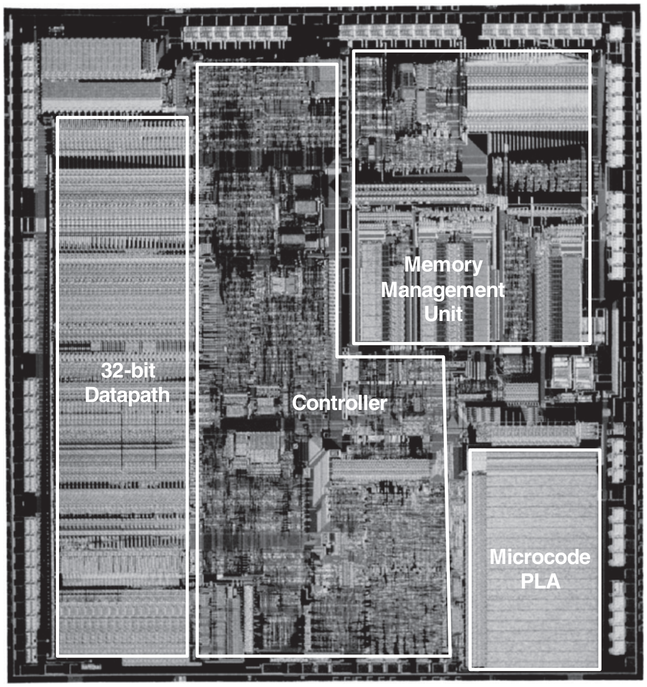


Figure 7.74 80386 microprocessor chip

clearly visible on the left. Each of the columns processes one bit of data. Some of the control signals are generated using a *microcode* PLA that steps through the various states of the control FSM. The memory management unit in the upper right controls access to the external memory.

The 80486, shown in Figure 7.75, dramatically improved performance using pipelining. The datapath is again clearly visible, along with the control logic and microcode PLA. The 80486 added an on-chip floating-point unit; previous Intel processors either sent floating-point instructions to a separate coprocessor or emulated them in software. The 80486 was too fast for external memory to keep up, so it incorporated an 8-KB cache onto the chip to hold the most commonly used instructions and data. Chapter 8 describes caches in more detail and revisits the cache systems on Intel x86 processors.

The Pentium processor, shown in Figure 7.76, is a superscalar processor capable of executing two instructions simultaneously. Intel switched to the name Pentium instead of 80586 because AMD was becoming a serious competitor selling interchangeable 80486 chips, and part numbers

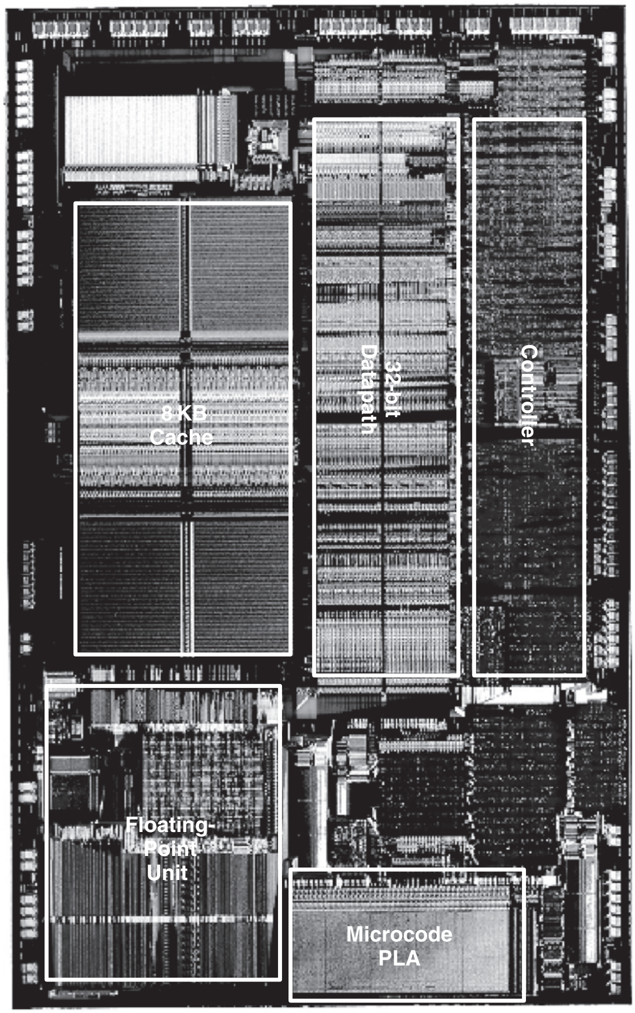


Figure 7.75 **80486 microprocessor chip**

cannot be trademarked. The Pentium uses separate instruction and data caches. It also uses a branch predictor to reduce the performance penalty for branches.

The Pentium Pro, Pentium II, and Pentium III processors all share a common out-of-order microarchitecture, code-named P6. The complex x86 instructions are broken down into one or more micro-ops similar to MIPS instructions. The micro-ops are then executed on a fast out-of-order execution core with an 11-stage pipeline. Figure 7.77 shows the Pentium III. The 32-bit datapath is called the Integer Execution Unit (IEU).

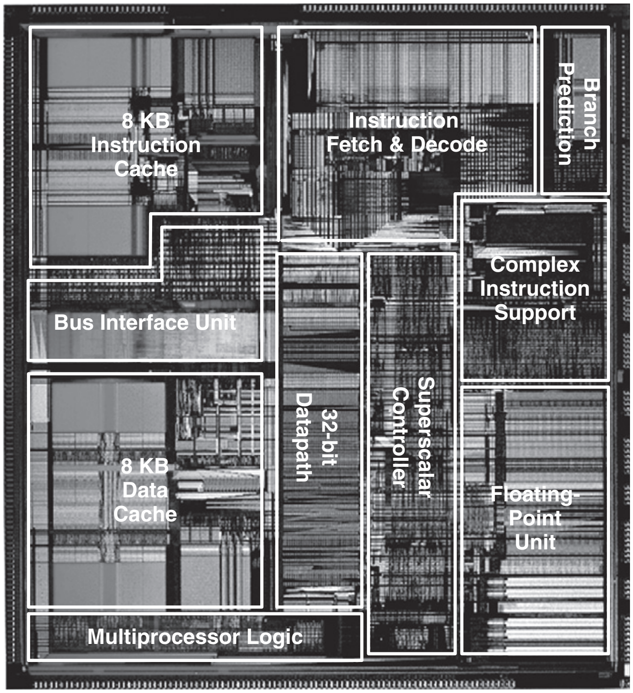

8 KB Instruction Cache
Instruction Fetch & Decode
Branch Prediction
Complex Instruction Support
Bus Interface Unit
32-bit Datapath
Superscalar Controller
8 KB Data Cache
Floating-Point Unit
Multiprocessor Logic

Figure 7.76 Pentium microprocessor chip

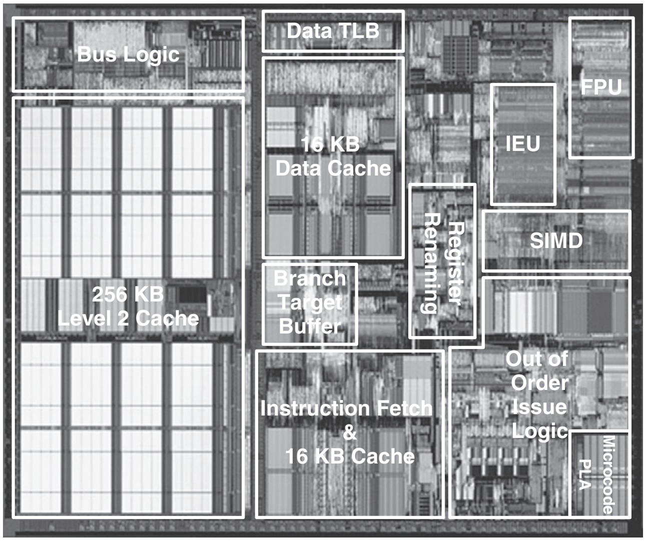

Bus Logic
Data TLB
FPU
16 KB Data Cache
IEU
Register Renaming
SIMD
Branch Target Buffer
256 KB Level 2 Cache
Out of Order Issue Logic
Instruction Fetch & 16 KB Cache
Microcode PLA

Figure 7.77 Pentium III microprocessor chip

The floating-point datapath is called the Floating Point Unit (FPU). The processor also has a SIMD unit to perform packed operations on short integer and floating-point data. A larger portion of the chip is dedicated to issuing instructions out-of-order than to actually executing the instructions. The instruction and data caches have grown to 16 KB each. The Pentium III also has a larger but slower 256-KB second-level cache on the same chip.

By the late 1990s, processors were marketed largely on clock speed. The Pentium 4 is another out-of-order processor with a very deep pipeline to achieve extremely high clock frequencies. It started with 20 stages, and later versions adopted 31 stages to achieve frequencies greater than 3 GHz. The chip, shown in Figure 7.78, packs in 42 to 178 million transistors (depending on the cache size), so even the major execution units are difficult to see on the photograph. Decoding three x86 instructions per cycle is impossible at such high clock frequencies because the instruction encodings are so complex and irregular. Instead, the processor predecodes the instructions into simpler micro-ops, then stores the micro-ops in a memory called a *trace cache*. Later versions of the Pentium 4 also perform multithreading to increase the throughput of multiple threads.

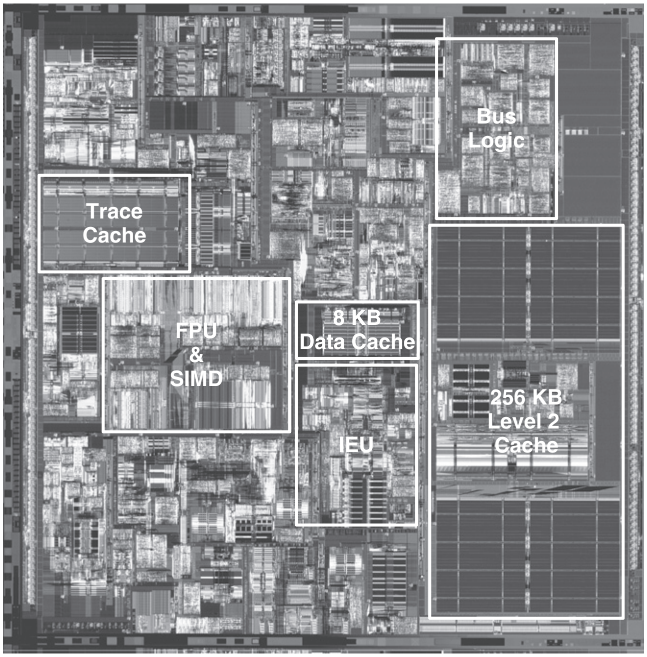


Figure 7.78 **Pentium 4 microprocessor chip**

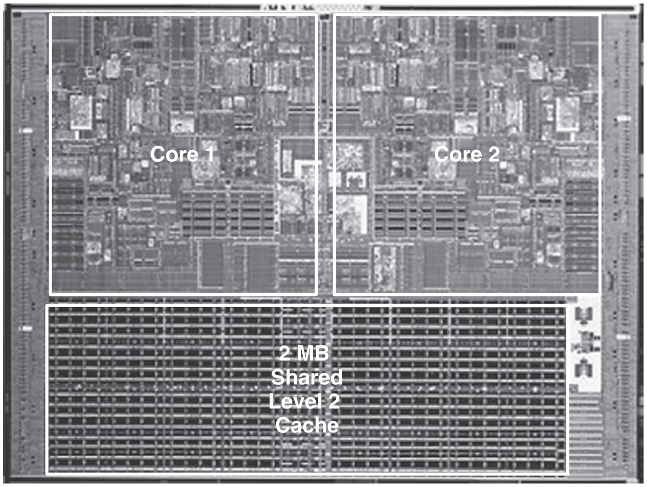


Figure 7.79 **Core Duo microprocessor chip**

The Pentium 4's reliance on deep pipelines and high clock speed led to extremely high power consumption, sometimes more than 100 W. This is unacceptable in laptops and makes cooling of desktops expensive.

Intel discovered that the older P6 architecture could achieve comparable performance at much lower clock speed and power. The Pentium M uses an enhanced version of the P6 out-of-order microarchitecture with 32-KB instruction and data caches and a 1- to 2-MB second-level cache. The Core Duo is a multicore processor based on two Pentium M cores connected to a shared 2-MB second-level cache. The individual functional units in Figure 7.79 are difficult to see, but the two cores and the large cache are clearly visible.

In 2009, Intel introduced a new microarchitecture code-named Nehalem that streamlines the initial Core microarchitecture. These processors, including the Core i3, i5, and i7 series, extend the instruction set to 64 bits. They offer from 2 to 6 cores, 3–15 MB of third-level cache, and a built-in memory controller. Some models include built-in graphics processors, also called *graphics accelerators*. Some support "Turbo-Boost" to improve the performance of single-threaded code by turning off the unused cores and raising the voltage and clock frequency of the boosted core. Some offer "hyperthreading," Intel's term for 2-way multithreading that doubles the number of cores from the user's perspective. Figure 7.80 shows a Core i7 die with four cores and 8 MB of shared L3 cache.

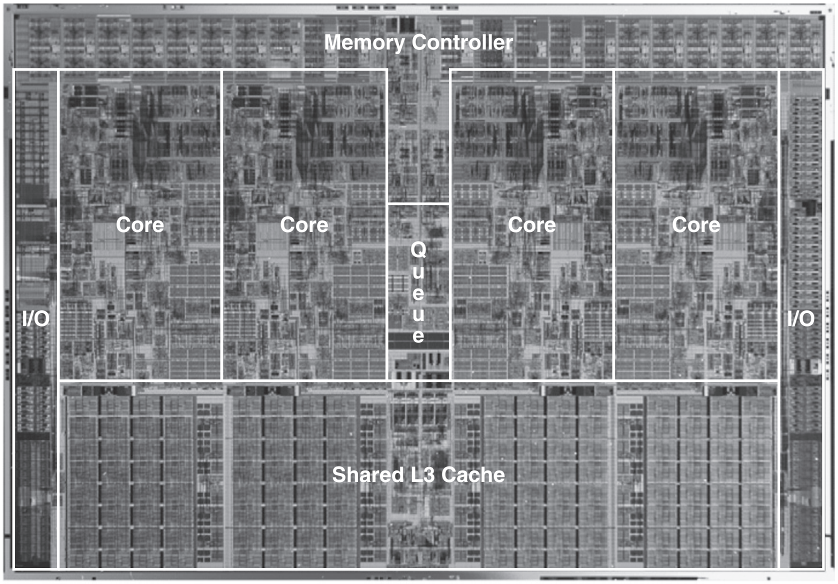


Figure 7.80 **Core i7 microprocessor chip**
(Source: http://www.intel.com/pressroom/archive/releases/2008/20081117comp_sm.htm. Courtesy Intel)

7.10 SUMMARY

This chapter has described three ways to build MIPS processors, each with different performance and cost trade-offs. We find this topic almost magical: how can such a seemingly complicated device as a microprocessor actually be simple enough to fit in a half-page schematic? Moreover, the inner workings, so mysterious to the uninitiated, are actually reasonably straightforward.

The MIPS microarchitectures have drawn together almost every topic covered in the text so far. Piecing together the microarchitecture puzzle illustrates the principles introduced in previous chapters, including the design of combinational and sequential circuits, covered in Chapters 2 and 3; the application of many of the building blocks described in Chapter 5; and the implementation of the MIPS architecture, introduced in Chapter 6. The MIPS microarchitectures can be described in a few pages of HDL, using the techniques from Chapter 4.

Building the microarchitectures has also heavily used our techniques for managing complexity. The microarchitectural abstraction forms the link between the logic and architecture abstractions, forming the crux of this book on digital design and computer architecture. We also use the abstractions of block diagrams and HDL to succinctly describe the arrangement of components. The microarchitectures exploit regularity and modularity, reusing a library of common building blocks such as ALUs, memories, multiplexers, and registers. Hierarchy is used in numerous ways. The microarchitectures are partitioned into the datapath and control units. Each of these units is built from logic blocks, which can be built from gates, which in turn can be built from transistors using the techniques developed in the first five chapters.

This chapter has compared single-cycle, multicycle, and pipelined microarchitectures for the MIPS processor. All three microarchitectures implement the same subset of the MIPS instruction set and have the same architectural state. The single-cycle processor is the most straightforward and has a CPI of 1.

The multicycle processor uses a variable number of shorter steps to execute instructions. It thus can reuse the ALU, rather than requiring several adders. However, it does require several nonarchitectural registers to store results between steps. The multicycle design in principle could be faster, because not all instructions must be equally long. In practice, it is generally slower, because it is limited by the slowest steps and by the sequencing overhead in each step.

The pipelined processor divides the single-cycle processor into five relatively fast pipeline stages. It adds pipeline registers between the stages to separate the five instructions that are simultaneously executing. It nominally has a CPI of 1, but hazards force stalls or flushes that increase the CPI slightly. Hazard resolution also costs some extra hardware and design complexity. The clock period ideally could be five times shorter than that of the single-cycle processor. In practice, it is not that short, because it is limited by the slowest stage and by the sequencing overhead in each stage. Nevertheless, pipelining provides substantial performance benefits. All modern high-performance microprocessors use pipelining today.

Although the microarchitectures in this chapter implement only a subset of the MIPS architecture, we have seen that supporting more instructions involves straightforward enhancements of the datapath and controller. Supporting exceptions also requires simple modifications.

A major limitation of this chapter is that we have assumed an ideal memory system that is fast and large enough to store the entire program and data. In reality, large fast memories are prohibitively expensive. The next chapter shows how to get most of the benefits of a large fast memory with a small fast memory that holds the most commonly used information and one or more larger but slower memories that hold the rest of the information.

Exercises

Exercise 7.1 Suppose that one of the following control signals in the single-cycle MIPS processor has a *stuck-at-0 fault*, meaning that the signal is always 0, regardless of its intended value. What instructions would malfunction? Why?

(a) *RegWrite*

(b) $ALUOp_1$

(c) *MemWrite*

Exercise 7.2 Repeat Exercise 7.1, assuming that the signal has a stuck-at-1 fault.

Exercise 7.3 Modify the single-cycle MIPS processor to implement one of the following instructions. See Appendix B for a definition of the instructions. Mark up a copy of Figure 7.11 to indicate the changes to the datapath. Name any new control signals. Mark up a copy of Table 7.8 to show the changes to the main decoder. Describe any other changes that are required.

(a) `sll`

(b) `lui`

(c) `slti`

(d) `blez`

Table 7.8 **Main decoder truth table to mark up with changes**

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01

Exercise 7.4 Repeat Exercise 7.3 for the following MIPS instructions.

(a) `jal`

(b) `lh`

(c) `jr`

(d) `srl`

Exercise 7.5 Many processor architectures have a *load with postincrement* instruction, which updates the index register to point to the next memory word after completing the load. `lwinc $rt, imm($rs)` is equivalent to the following two instructions:

```
lw $rt, imm($rs)
addi $rs, $rs, 4
```

Repeat Exercise 7.3 for the `lwinc` instruction. Is it possible to add the instruction without modifying the register file?

Exercise 7.6 Add a single-precision floating-point unit to the single-cycle MIPS processor to handle `add.s`, `sub.s`, and `mul.s`. Assume that you have single-precision floating-point adder and multiplier units available. Explain what changes must be made to the datapath and the controller.

Exercise 7.7 Your friend is a crack circuit designer. She has offered to redesign one of the units in the single-cycle MIPS processor to have half the delay. Using the delays from Table 7.6, which unit should she work on to obtain the greatest speedup of the overall processor, and what would the cycle time of the improved machine be?

Exercise 7.8 Consider the delays given in Table 7.6. Ben Bitdiddle builds a prefix adder that reduces the ALU delay by 20 ps. If the other element delays stay the same, find the new cycle time of the single-cycle MIPS processor and determine how long it takes to execute a benchmark with 100 billion instructions.

Exercise 7.9 Suppose one of the following control signals in the multicycle MIPS processor has a stuck-at-0 fault, meaning that the signal is always 0, regardless of its intended value. What instructions would malfunction? Why?

(a) *MemtoReg*

(b) $ALUOp_0$

(c) *PCSrc*

Exercise 7.10 Repeat Exercise 7.9, assuming that the signal has a stuck-at-1 fault.

Exercise 7.11 Modify the HDL code for the single-cycle MIPS processor, given in Section 7.6.1, to handle one of the new instructions from Exercise 7.3. Enhance the testbench, given in Section 7.6.3, to test the new instruction.

Exercise 7.12 Repeat Exercise 7.11 for the new instructions from Exercise 7.4.

Exercise 7.13 Modify the multicycle MIPS processor to implement one of the following instructions. See Appendix B for a definition of the instructions. Mark up a copy of Figure 7.27 to indicate the changes to the datapath. Name any new control signals. Mark up a copy of Figure 7.39 to show the changes to the controller FSM. Describe any other changes that are required.

(a) `srlv`

(b) `ori`

(c) `xori`

(d) `jr`

Exercise 7.14 Repeat Exercise 7.13 for the following MIPS instructions.

(a) `bne`

(b) `lb`

(c) `lbu`

(d) `andi`

Exercise 7.15 Repeat Exercise 7.5 for the multicycle MIPS processor. Show the changes to the multicycle datapath and control FSM. Is it possible to add the instruction without modifying the register file?

Exercise 7.16 Repeat Exercise 7.6 for the multicycle MIPS processor.

Exercise 7.17 Suppose that the floating-point adder and multiplier from Exercise 7.16 each take two cycles to operate. In other words, the inputs are applied at the beginning of one cycle, and the output is available in the second cycle. How does your answer to Exercise 7.16 change?

Exercise 7.18 Your friend, the crack circuit designer, has offered to redesign one of the units in the multicycle MIPS processor to be much faster. Using the delays from Table 7.6, which unit should she work on to obtain the greatest speedup of the overall processor? How fast should it be? (Making it faster than necessary is a waste of your friend's effort.) What is the cycle time of the improved processor?

Exercise 7.19 Repeat Exercise 7.8 for the multicycle processor. Assume the instruction mix of Example 7.7.

Exercise 7.20 Suppose the multicycle MIPS processor has the component delays given in Table 7.6. Alyssa P. Hacker designs a new register file that has 40% less power but twice as much delay. Should she switch to the slower but lower power register file for her multicycle processor design?

Exercise 7.21 Goliath Corp claims to have a patent on a three-ported register file. Rather than fighting Goliath in court, Ben Bitdiddle designs a new register file that has only a single read/write port (like the combined instruction and data memory). Redesign the MIPS multicycle datapath and controller to use his new register file.

Exercise 7.22 What is the CPI of the redesigned multicycle MIPS processor from Exercise 7.21? Use the instruction mix from Example 7.7.

Exercise 7.23 How many cycles are required to run the following program on the multicycle MIPS processor? What is the CPI of this program?

```
   addi $s0, $0, done  # result = 5

while:
   beq $s0, $0, done   # if result > 0, execute while block
   addi $s0, $s0, -1   # while block: result = result-1
   j    while

done:
```

Exercise 7.24 Repeat Exercise 7.23 for the following program.

```
   add   $s0, $0,  $0     # i = 0
   add   $s1, $0,  $0     # sum = 0
   addi  $t0, $0,  10     # $t0 = 10

loop:
   slt   $t1, $s0, $t0    # if (i < 10), $t1 = 1, else $t1 = 0
   beq   $t1, $0,  done   # if $t1 == 0 (i >= 10), branch to done
   add   $s1, $s1, $s0    # sum = sum + i
   addi  $s0, $s0, 1      # increment i
   j loop
done:
```

Exercise 7.25 Write HDL code for the multicycle MIPS processor. The processor should be compatible with the following top-level module. The `mem` module is used to hold both instructions and data. Test your processor using the testbench from Section 7.6.3.

```
module top(input  logic        clk, reset,
           output logic [31:0] writedata, adr,
           output logic        memwrite);

  logic [31:0] readdata;

  // instantiate processor and memories
  mips mips(clk, reset, adr, writedata, memwrite, readdata);
  mem mem(clk, memwrite, adr, writedata, readdata);

endmodule

module mem(input  logic        clk, we,
           input  logic [31:0] a, wd,
           output logic [31:0] rd);

   logic [31:0] RAM[63:0];

initial
   begin
    $readmemh("memfile.dat", RAM);
   end
```

```
    assign rd = RAM[a[31:2]]; // word aligned
    always @(posedge clk)
      if (we)
       RAM[a[31:2]] <= wd;
  endmodule
```

Exercise 7.26 Extend your HDL code for the multicycle MIPS processor from Exercise 7.25 to handle one of the new instructions from Exercise 7.13. Enhance the testbench to test the new instruction.

Exercise 7.27 Repeat Exercise 7.26 for one of the new instructions from Exercise 7.14.

Exercise 7.28 The pipelined MIPS processor is running the following program. Which registers are being written, and which are being read on the fifth cycle?

```
addi $s1, $s2, 5
sub  $t0, $t1, $t2
lw   $t3, 15($s1)
sw   $t5, 72($t0)
or   $t2, $s4, $s5
```

Exercise 7.29 Repeat Exercise 7.28 for the following MIPS program. Recall that the pipelined MIPS processor has a hazard unit.

```
add  $s0, $t0, $t1
sub  $s1, $t2, $t3
and  $s2, $s0, $s1
or   $s3, $t4, $t5
slt  $s4, $s2, $s3
```

Exercise 7.30 Using a diagram similar to Figure 7.52, show the forwarding and stalls needed to execute the following instructions on the pipelined MIPS processor.

```
add  $t0, $s0, $s1
sub  $t0, $t0, $s2
Iw   $t1, 60($t0)
and  $t2, $t1, $t0
```

Exercise 7.31 Repeat Exercise 7.30 for the following instructions.

```
add  $t0, $s0, $s1
Iw   $t1, 60($s2)
sub  $t2, $t0, $s3
and  $t3, $t1, $t0
```

Exercise 7.32 How many cycles are required for the pipelined MIPS processor to issue all of the instructions for the program in Exercise 7.23? What is the CPI of the processor on this program?

Exercise 7.33 Repeat Exercise 7.32 for the instructions of the program in Exercise 7.24.

Exercise 7.34 Explain how to extend the pipelined MIPS processor to handle the `addi` instruction.

Exercise 7.35 Explain how to extend the pipelined processor to handle the `j` instruction. Give particular attention to how the pipeline is flushed when a jump takes place.

Exercise 7.36 Examples 7.9 and 7.10 point out that the pipelined MIPS processor performance might be better if branches take place during the Execute stage rather than the Decode stage. Show how to modify the pipelined processor from Figure 7.58 to branch in the Execute stage. How do the stall and flush signals change? Redo Examples 7.9 and 7.10 to find the new CPI, cycle time, and overall time to execute the program.

Exercise 7.37 Your friend, the crack circuit designer, has offered to redesign one of the units in the pipelined MIPS processor to be much faster. Using the delays from Table 7.6 and Example 7.10, which unit should she work on to obtain the greatest speedup of the overall processor? How fast should it be? (Making it faster than necessary is a waste of your friend's effort.) What is the cycle time of the improved processor?

Exercise 7.38 Consider the delays from Table 7.6 and Example 7.10. Now suppose that the ALU were 20% faster. Would the cycle time of the pipelined MIPS processor change? What if the ALU were 20% slower?

Exercise 7.39 Suppose the MIPS pipelined processor is divided into 10 stages of 400 ps each, including sequencing overhead. Assume the instruction mix of Example 7.7. Also assume that 50% of the loads are immediately followed by an instruction that uses the result, requiring six stalls, and that 30% of the branches are mispredicted. The target address of a branch or jump instruction is not computed until the end of the second stage. Calculate the average CPI and execution time of computing 100 billion instructions from the SPECINT2000 benchmark for this 10-stage pipelined processor.

Exercise 7.40 Write HDL code for the pipelined MIPS processor. The processor should be compatible with the top-level module from HDL Example 7.13. It should support all of the instructions described in this chapter, including `addi` and `j` (see Exercises 7.34 and 7.35). Test your design using the testbench from HDL Example 7.12.

Exercise 7.41 Design the hazard unit shown in Figure 7.58 for the pipelined MIPS processor. Use an HDL to implement your design. Sketch the hardware that a synthesis tool might generate from your HDL.

Exercise 7.42 A *nonmaskable interrupt* (*NMI*) is triggered by an input pin to the processor. When the pin is asserted, the current instruction should finish, then the processor should set the Cause register to 0 and take an exception. Show how to modify the multicycle processor in Figures 7.63 and 7.64 to handle nonmaskable interrupts.

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 7.1 Explain the advantages of pipelined microprocessors.

Question 7.2 If additional pipeline stages allow a processor to go faster, why don't processors have 100 pipeline stages?

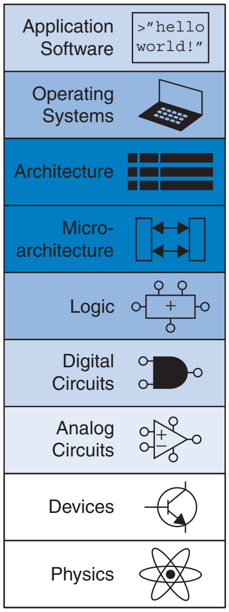
Question 7.3 Describe what a hazard is in a microprocessor and explain ways in which it can be resolved. What are the pros and cons of each way?

Question 7.4 Describe the concept of a superscalar processor and its pros and cons.

Memory and I/O Systems

8

8.1 Introduction
8.2 Memory System Performance Analysis
8.3 Caches
8.4 Virtual Memory
8.5 I/O Introduction
8.6 Embedded I/O Systems
8.7 PC I/O Systems
8.8 Real-World Perspective: x86 Memory and I/O Systems*
8.9 Summary
Epilogue
Exercises
Interview Questions



8.1 INTRODUCTION

A computer's ability to solve problems is influenced by its memory system and the input/output (I/O) devices – such as monitors, keyboards, and printers – that allow us to manipulate and view the results of its computations. This chapter investigates these practical memory and I/O systems.

Computer system performance depends on the memory system as well as the processor microarchitecture. Chapter 7 assumed an ideal memory system that could be accessed in a single clock cycle. However, this would be true only for a very small memory—or a very slow processor! Early processors were relatively slow, so memory was able to keep up. But processor speed has increased at a faster rate than memory speeds. DRAM memories are currently 10 to 100 times slower than processors. The increasing gap between processor and DRAM memory speeds demands increasingly ingenious memory systems to try to approximate a memory that is as fast as the processor. The first half of this chapter investigates memory systems and considers trade-offs of speed, capacity, and cost.

The processor communicates with the memory system over a *memory interface*. Figure 8.1 shows the simple memory interface used in our multicycle MIPS processor. The processor sends an address over the *Address* bus to the memory system. For a read, *MemWrite* is 0 and the memory returns the data on the *ReadData* bus. For a write, *MemWrite* is 1 and the processor sends data to memory on the *WriteData* bus.

The major issues in memory system design can be broadly explained using a metaphor of books in a library. A library contains many books on the shelves. If you were writing a term paper on the meaning of dreams, you might go to the library[1] and pull Freud's *The Interpretation of Dreams*

---

[1] We realize that library usage is plummeting among college students because of the Internet. But we also believe that libraries contain vast troves of hard-won human knowledge that are not electronically available. We hope that Web searching does not completely displace the art of library research.

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00008-2
© 2013 Elsevier, Inc. All rights reserved.

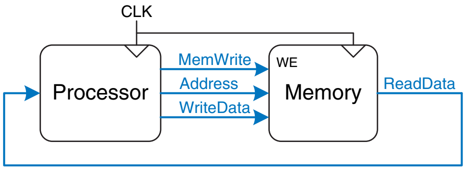


Figure 8.1 **The memory interface**

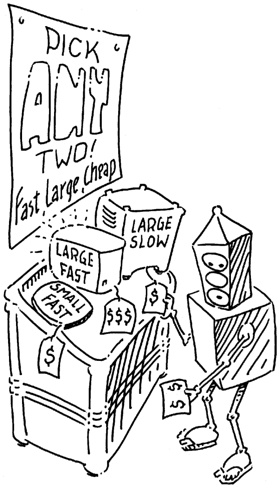
off the shelf and bring it to your cubicle. After skimming it, you might put it back and pull out Jung's *The Psychology of the Unconscious*. You might then go back for another quote from *Interpretation of Dreams*, followed by yet another trip to the stacks for Freud's *The Ego and the Id*. Pretty soon you would get tired of walking from your cubicle to the stacks. If you are clever, you would save time by keeping the books in your cubicle rather than schlepping them back and forth. Furthermore, when you pull a book by Freud, you could also pull several of his other books from the same shelf.

This metaphor emphasizes the principle, introduced in Section 6.2.1, of making the common case fast. By keeping books that you have recently used or might likely use in the future at your cubicle, you reduce the number of time-consuming trips to the stacks. In particular, you use the principles of *temporal* and *spatial locality*. Temporal locality means that if you have used a book recently, you are likely to use it again soon. Spatial locality means that when you use one particular book, you are likely to be interested in other books on the same shelf.

The library itself makes the common case fast by using these principles of locality. The library has neither the shelf space nor the budget to accommodate all of the books in the world. Instead, it keeps some of the lesser-used books in deep storage in the basement. Also, it may have an interlibrary loan agreement with nearby libraries so that it can offer more books than it physically carries.

In summary, you obtain the benefits of both a large collection and quick access to the most commonly used books through a hierarchy of storage. The most commonly used books are in your cubicle. A larger collection is on the shelves. And an even larger collection is available, with advanced notice, from the basement and other libraries. Similarly, memory systems use a hierarchy of storage to quickly access the most commonly used data while still having the capacity to store large amounts of data.

Memory subsystems used to build this hierarchy were introduced in Section 5.5. Computer memories are primarily built from dynamic RAM (DRAM) and static RAM (SRAM). Ideally, the computer memory system is fast, large, and cheap. In practice, a single memory only has two of these three attributes; it is either slow, small, or expensive. But computer systems can approximate the ideal by combining a fast small cheap memory and a slow large cheap memory. The fast memory stores the most commonly used data and instructions, so on average the memory system appears fast. The



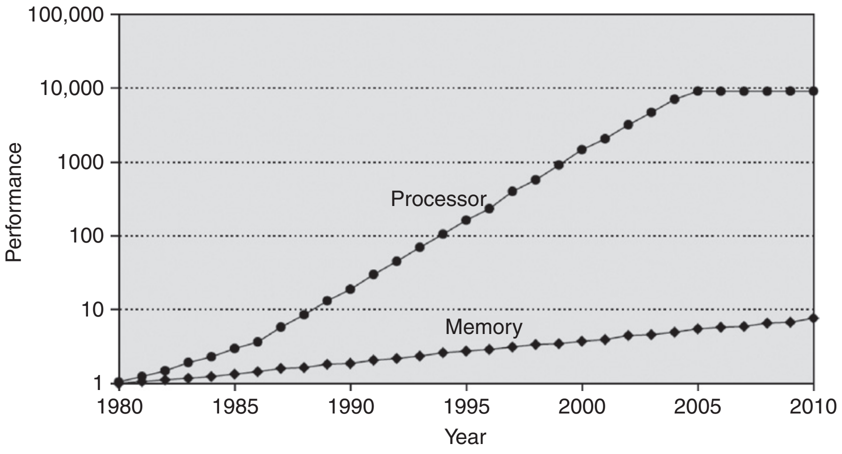


Figure 8.2 Diverging processor and memory performance Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012.

large memory stores the remainder of the data and instructions, so the overall capacity is large. The combination of two cheap memories is much less expensive than a single large fast memory. These principles extend to using an entire hierarchy of memories of increasing capacity and decreasing speed.

Computer memory is generally built from DRAM chips. In 2012, a typical PC had a *main memory* consisting of 4 to 8 GB of DRAM, and DRAM cost about $10 per gigabyte (GB). DRAM prices have declined at about 25% per year for the last three decades, and memory capacity has grown at the same rate, so the total cost of the memory in a PC has remained roughly constant. Unfortunately, DRAM speed has improved by only about 7% per year, whereas processor performance has improved at a rate of 25 to 50% per year, as shown in Figure 8.2. The plot shows memory (DRAM) and processor speeds with the 1980 speeds as a baseline. In about 1980, processor and memory speeds were the same. But performance has diverged since then, with memories badly lagging.[2]

DRAM could keep up with processors in the 1970s and early 1980's, but it is now woefully too slow. The DRAM access time is one to two orders of magnitude longer than the processor cycle time (tens of nanoseconds, compared to less than one nanosecond).

To counteract this trend, computers store the most commonly used instructions and data in a faster but smaller memory, called a *cache*. The cache is usually built out of SRAM on the same chip as the processor. The cache speed is comparable to the processor speed, because SRAM is inherently faster than DRAM, and because the on-chip memory eliminates lengthy delays caused by traveling to and from a separate chip. In 2012, on-chip SRAM costs were on the order of $10,000/GB, but the cache is relatively small (kilobytes to several megabytes), so the overall cost is low. Caches can store both instructions and data, but we will refer to their contents generically as "data."

---

[2] Although recent single processor performance has remained approximately constant, as shown in Figure 8.2 for the years 2005–2010, the increase in multi-core systems (not depicted on the graph) only worsens the gap between processor and memory performance.

If the processor requests data that is available in the cache, it is returned quickly. This is called a cache *hit*. Otherwise, the processor retrieves the data from main memory (DRAM). This is called a cache *miss*. If the cache hits most of the time, then the processor seldom has to wait for the slow main memory, and the average access time is low.

The third level in the memory hierarchy is the hard drive. In the same way that a library uses the basement to store books that do not fit in the stacks, computer systems use the hard drive to store data that does not fit in main memory. In 2012, a hard disk drive (HDD), built using magnetic storage, cost less than $0.10/GB and had an access time of about 10 ms. Hard disk costs have decreased at 60%/year but access times scarcely improved. Solid state drives (SSDs), built using flash memory technology, are an increasingly common alternative to HDDs. SSDs have been used by niche markets for over two decades, and they were introduced into the mainstream market in 2007. SSDs overcome some of the mechanical failures of HDDs, but they cost ten times as much at $1/GB.

The hard drive provides an illusion of more capacity than actually exists in the main memory. It is thus called virtual memory. Like books in the basement, data in virtual memory takes a long time to access. Main memory, also called physical memory, holds a subset of the virtual memory. Hence, the main memory can be viewed as a cache for the most commonly used data from the hard drive.

Figure 8.3 summarizes the memory hierarchy of the computer system discussed in the rest of this chapter. The processor first seeks data in a small but fast cache that is usually located on the same chip. If the data is not available in the cache, the processor then looks in main memory. If the data is not there either, the processor fetches the data from virtual memory on the large but slow hard disk. Figure 8.4 illustrates this capacity and speed trade-off in the memory hierarchy and lists typical costs, access times, and bandwidth in 2012 technology. As access time decreases, speed increases.

Section 8.2 introduces memory system performance analysis. Section 8.3 explores several cache organizations, and Section 8.4 delves into virtual memory systems. To conclude, this chapter explores how processors can

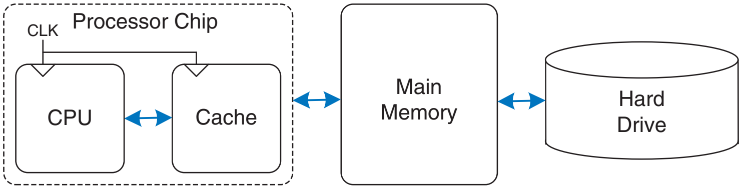
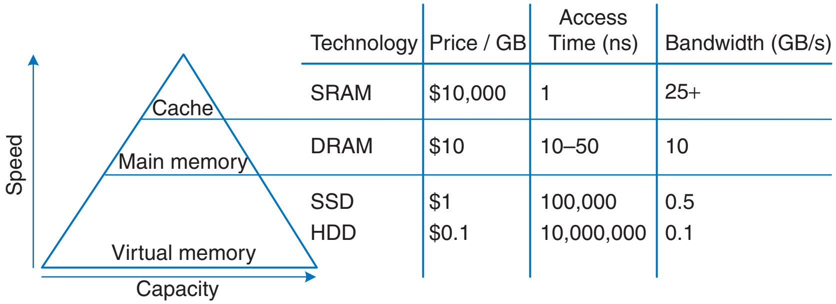


Figure 8.3 A typical memory hierarchy



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10–50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

Figure 8.4 **Memory hierarchy components, with typical characteristics in 2012**

access input and output devices, such as keyboards and monitors, in much the same way as they access memory. Section 8.5 investigates such memory-mapped I/O. Section 8.6 addresses I/O for embedded systems, and Section 8.7 describes major I/O standards for personal computers.

8.2 MEMORY SYSTEM PERFORMANCE ANALYSIS

Designers (and computer buyers) need quantitative ways to measure the performance of memory systems to evaluate the cost-benefit trade-offs of various alternatives. Memory system performance metrics are *miss rate* or *hit rate* and *average memory access time.* Miss and hit rates are calculated as:

$$\begin{aligned} Miss\ Rate &= \frac{Number\ of\ misses}{Number\ of\ total\ memory\ accesses} = 1 - Hit\ Rate \\ Hit\ Rate &= \frac{Number\ of\ hits}{Number\ of\ total\ memory\ accesses} = 1 - Miss\ Rate \end{aligned} \tag{8.1}$$

Example 8.1 CALCULATING CACHE PERFORMANCE

Suppose a program has 2000 data access instructions (loads or stores), and 1250 of these requested data values are found in the cache. The other 750 data values are supplied to the processor by main memory or disk memory. What are the miss and hit rates for the cache?

Solution: The miss rate is 750/2000 = 0.375 = 37.5%. The hit rate is 1250/2000 = 0.625 = 1 − 0.375 = 62.5%.

Average memory access time (*AMAT*) is the average time a processor must wait for memory per load or store instruction. In the typical computer system from Figure 8.3, the processor first looks for the data in the cache. If the cache misses, the processor then looks in main memory. If the main memory misses, the processor accesses virtual memory on the hard disk. Thus, *AMAT* is calculated as:

$$AMAT = t_{\text{cache}} + MR_{\text{cache}}(t_{MM} + MR_{MM}t_{VM}) \tag{8.2}$$

where t_{cache}, t_{MM}, and t_{VM} are the access times of the cache, main memory, and virtual memory, and MR_{cache} and MR_{MM} are the cache and main memory miss rates, respectively.

Example 8.2 CALCULATING AVERAGE MEMORY ACCESS TIME

Suppose a computer system has a memory organization with only two levels of hierarchy, a cache and main memory. What is the average memory access time given the access times and miss rates in Table 8.1?

Solution: The average memory access time is $1 + 0.1(100) = 11$ cycles.

Table 8.1 Access times and miss rates

Memory Level	Access Time (Cycles)	Miss Rate
Cache	1	10%
Main Memory	100	0%

Example 8.3 IMPROVING ACCESS TIME

An 11-cycle average memory access time means that the processor spends ten cycles waiting for data for every one cycle actually using that data. What cache miss rate is needed to reduce the average memory access time to 1.5 cycles given the access times in Table 8.1?

Solution: If the miss rate is m, the average access time is $1 + 100m$. Setting this time to 1.5 and solving for m requires a cache miss rate of 0.5%.

As a word of caution, performance improvements might not always be as good as they sound. For example, making the memory system ten times faster will not necessarily make a computer program run ten times as fast. If 50% of a program's instructions are loads and stores, a tenfold memory system improvement only means a 1.82-fold improvement in program performance. This general principle is called *Amdahl's Law*, which says that the effort spent on increasing the performance of a subsystem is worthwhile only if the subsystem affects a large percentage of the overall performance.

Gene Amdahl, 1922–. Most famous for Amdahl's Law, an observation he made in 1965. While in graduate school, he began designing computers in his free time. This side work earned him his Ph.D. in theoretical physics in 1952. He joined IBM immediately after graduation, and later went on to found three companies, including one called Amdahl Corporation in 1970.

8.3 CACHES

A cache holds commonly used memory data. The number of data words that it can hold is called the *capacity*, C. Because the capacity of the cache is smaller than that of main memory, the computer system designer must choose what subset of the main memory is kept in the cache.

When the processor attempts to access data, it first checks the cache for the data. If the cache hits, the data is available immediately. If the cache misses, the processor fetches the data from main memory and places it in the cache for future use. To accommodate the new data, the cache must *replace* old data. This section investigates these issues in cache design by answering the following questions: (1) What data is held in the cache? (2) How is data found? and (3) What data is replaced to make room for new data when the cache is full?

When reading the next sections, keep in mind that the driving force in answering these questions is the inherent spatial and temporal locality of data accesses in most applications. Caches use spatial and temporal locality to predict what data will be needed next. If a program accesses data in a random order, it would not benefit from a cache.

As we explain in the following sections, caches are specified by their capacity (C), number of sets (S), block size (b), number of blocks (B), and degree of associativity (N).

Although we focus on data cache loads, the same principles apply for fetches from an instruction cache. Data cache store operations are similar and are discussed further in Section 8.3.4.

Cache: a hiding place especially for concealing and preserving provisions or implements.
 – Merriam Webster
Online Dictionary. 2012.
www.merriam-webster.com

8.3.1 What Data is Held in the Cache?

An ideal cache would anticipate all of the data needed by the processor and fetch it from main memory ahead of time so that the cache has a zero miss rate. Because it is impossible to predict the future with perfect accuracy, the cache must guess what data will be needed based on the past pattern of memory accesses. In particular, the cache exploits temporal and spatial locality to achieve a low miss rate.

Recall that temporal locality means that the processor is likely to access a piece of data again soon if it has accessed that data recently. Therefore, when the processor loads or stores data that is not in the cache, the data is copied from main memory into the cache. Subsequent requests for that data hit in the cache.

Recall that spatial locality means that, when the processor accesses a piece of data, it is also likely to access data in nearby memory locations. Therefore, when the cache fetches one word from memory, it may also fetch several adjacent words. This group of words is called a *cache block* or *cache line*. The number of words in the cache block, b, is called the *block size*. A cache of capacity C contains $B = C/b$ blocks.

The principles of temporal and spatial locality have been experimentally verified in real programs. If a variable is used in a program, the same variable is likely to be used again, creating temporal locality. If an element in an array is used, other elements in the same array are also likely to be used, creating spatial locality.

8.3.2 How is Data Found?

A cache is organized into S *sets*, each of which holds one or more blocks of data. The relationship between the address of data in main memory and the location of that data in the cache is called the *mapping*. Each memory address maps to exactly one set in the cache. Some of the address bits are used to determine which cache set contains the data. If the set contains more than one block, the data may be kept in any of the blocks in the set.

Caches are categorized based on the number of blocks in a set. In a *direct mapped* cache, each set contains exactly one block, so the cache has $S = B$ sets. Thus, a particular main memory address maps to a unique block in the cache. In an *N-way set associative* cache, each set contains N blocks. The address still maps to a unique set, with $S = B/N$ sets. But the data from that address can go in any of the N blocks in that set. A *fully associative* cache has only $S = 1$ set. Data can go in any of the B blocks in the set. Hence, a fully associative cache is another name for a B-way set associative cache.

To illustrate these cache organizations, we will consider a MIPS memory system with 32-bit addresses and 32-bit words. The memory is byte-addressable, and each word is four bytes, so the memory consists of 2^{30} words aligned on word boundaries. We analyze caches with an eight-word capacity (C) for the sake of simplicity. We begin with a one-word block size (b), then generalize later to larger blocks.

Direct Mapped Cache

A *direct mapped* cache has one block in each set, so it is organized into $S = B$ sets. To understand the mapping of memory addresses onto cache blocks, imagine main memory as being mapped into b-word blocks, just as the cache is. An address in block 0 of main memory maps to set 0 of the cache. An address in block 1 of main memory maps to set 1 of the cache, and so forth until an address in block $B - 1$ of main memory maps to block $B - 1$ of the cache. There are no more blocks of the cache, so the mapping wraps around, such that block B of main memory maps to block 0 of the cache.

This mapping is illustrated in Figure 8.5 for a direct mapped cache with a capacity of eight words and a block size of one word. The cache has eight sets, each of which contains a one-word block. The bottom two bits of the address are always 00, because they are word aligned. The next $\log_2 8 = 3$ bits indicate the set onto which the memory address maps. Thus, the data at addresses 0x00000004, 0x00000024, . . ., 0xFFFFFFE4 all

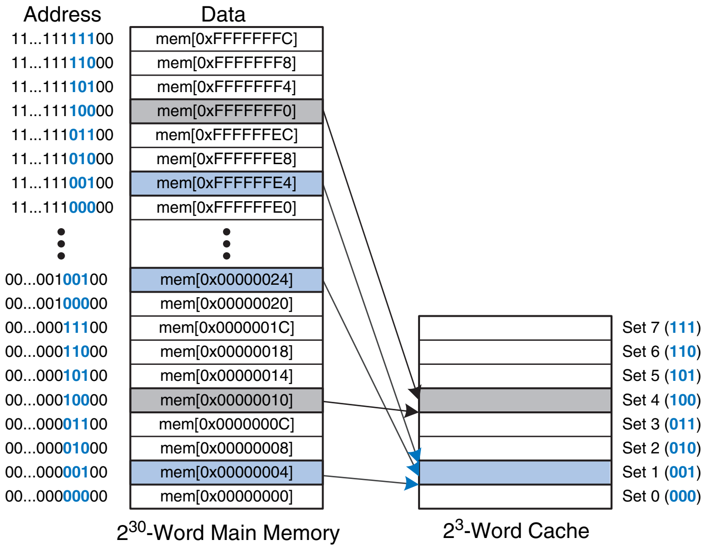

Address
Data
11...11111100
11...11111000
11...11110100
11...11110000
11...11101100
11...11101000
11...11100100
11...11100000
00...00100100
00...00100000
00...00011100
00...00011000
00...00010100
00...00010000
00...00001100
00...00001000
00...00000100
00...00000000
mem[0xFFFFFFFC]
mem[0xFFFFFFF8]
mem[0xFFFFFFF4]
mem[0xFFFFFFF0]
mem[0xFFFFFFEC]
mem[0xFFFFFFE8]
mem[0xFFFFFFE4]
mem[0xFFFFFFE0]
mem[0x00000024]
mem[0x00000020]
mem[0x0000001C]
mem[0x00000018]
mem[0x00000014]
mem[0x00000010]
mem[0x0000000C]
mem[0x00000008]
mem[0x00000004]
mem[0x00000000]
Set 7 (111)
Set 6 (110)
Set 5 (101)
Set 4 (100)
Set 3 (011)
Set 2 (010)
Set 1 (001)
Set 0 (000)
2^30-Word Main Memory
2^3-Word Cache

Figure 8.5 **Mapping of main memory to a direct mapped cache**

map to set 1, as shown in blue. Likewise, data at addresses 0x00000010, . . . , 0xFFFFFFF0 all map to set 4, and so forth. Each main memory address maps to exactly one set in the cache.

Example 8.4 CACHE FIELDS

To what cache set in Figure 8.5 does the word at address 0x00000014 map? Name another address that maps to the same set.

Solution: The two least significant bits of the address are 00, because the address is word aligned. The next three bits are 101, so the word maps to set 5. Words at addresses 0x34, 0x54, 0x74, . . . , 0xFFFFFFF4 all map to this same set.

Because many addresses map to a single set, the cache must also keep track of the address of the data actually contained in each set. The least significant bits of the address specify which set holds the data. The remaining most significant bits are called the *tag* and indicate which of the many possible addresses is held in that set.

In our previous examples, the two least significant bits of the 32-bit address are called the *byte offset*, because they indicate the byte within the word. The next three bits are called the *set bits*, because they indicate the set to which the address maps. (In general, the number of set bits is $\log_2 S$.) The remaining 27 tag bits indicate the memory address of the data stored in a given cache set. Figure 8.6 shows the cache fields for address 0xFFFFFFE4. It maps to set 1 and its tag is all 1's.

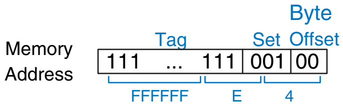

Memory Address
Tag
Set
Byte Offset
111 ... 111
001
00
FFFFFF
E
4

Figure 8.6 Cache fields for address 0xFFFFFFE4 when mapping to the cache in Figure 8.5

Example 8.5 CACHE FIELDS

Find the number of set and tag bits for a direct mapped cache with 1024 (2^{10}) sets and a one-word block size. The address size is 32 bits.

Solution: A cache with 2^{10} sets requires $\log_2(2^{10}) = 10$ set bits. The two least significant bits of the address are the byte offset, and the remaining $32 - 10 - 2 = 20$ bits form the tag.

Sometimes, such as when the computer first starts up, the cache sets contain no data at all. The cache uses a *valid bit* for each set to indicate whether the set holds meaningful data. If the valid bit is 0, the contents are meaningless.

Figure 8.7 shows the hardware for the direct mapped cache of Figure 8.5. The cache is constructed as an eight-entry SRAM. Each entry, or set, contains one line consisting of 32 bits of data, 27 bits of tag, and 1 valid bit. The cache is accessed using the 32-bit address. The two least significant bits, the byte offset bits, are ignored for word accesses. The next three bits, the set bits, specify the entry or set in the cache. A load instruction reads the specified entry from the cache and checks the tag and valid bits. If the tag matches the most significant 27 bits of the

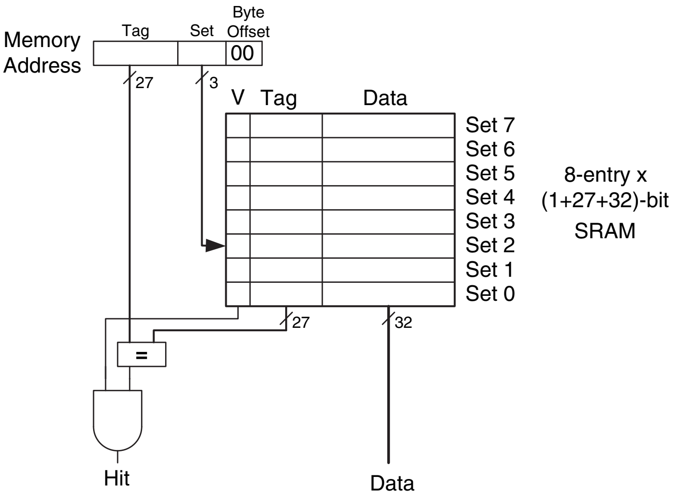

Memory Address
Tag
Set
Byte Offset
00
27
3
V
Tag
Data
Set 7
Set 6
Set 5
Set 4
Set 3
Set 2
Set 1
Set 0
8-entry x (1+27+32)-bit SRAM
27
32
=
Hit
Data

Figure 8.7 Direct mapped cache with 8 sets

address and the valid bit is 1, the cache hits and the data is returned to the processor. Otherwise, the cache misses and the memory system must fetch the data from main memory.

Example 8.6 TEMPORAL LOCALITY WITH A DIRECT MAPPED CACHE

Loops are a common source of temporal and spatial locality in applications. Using the eight-entry cache of Figure 8.7, show the contents of the cache after executing the following silly loop in MIPS assembly code. Assume that the cache is initially empty. What is the miss rate?

```
      addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0xC($0)
      lw   $t3, 0x8($0)
      addi $t0, $t0, −1
      j    loop
done:
```

Solution: The program contains a loop that repeats for five iterations. Each iteration involves three memory accesses (loads), resulting in 15 total memory accesses. The first time the loop executes, the cache is empty and the data must be fetched from main memory locations 0x4, 0xC, and 0x8 into cache sets 1, 3, and 2, respectively. However, the next four times the loop executes, the data is found in the cache. Figure 8.8 shows the contents of the cache during the last request to memory address 0x4. The tags are all 0 because the upper 27 bits of the addresses are 0. The miss rate is 3/15 = 20%.

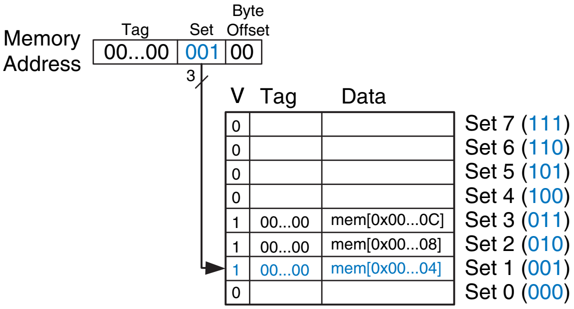

Memory Address
Tag
Set
Byte Offset
00...00
001
00
3
V
Tag
Data
0
0
0
0
1 00...00 mem[0x00...0C]
1 00...00 mem[0x00...08]
1 00...00 mem[0x00...04]
0
Set 7 (111)
Set 6 (110)
Set 5 (101)
Set 4 (100)
Set 3 (011)
Set 2 (010)
Set 1 (001)
Set 0 (000)

Figure 8.8 Direct mapped cache contents

When two recently accessed addresses map to the same cache block, a *conflict* occurs, and the most recently accessed address *evicts* the previous one from the block. Direct mapped caches have only one block in each set, so two addresses that map to the same set always cause a conflict. Example 8.7 on the next page illustrates conflicts.

Example 8.7 CACHE BLOCK CONFLICT

What is the miss rate when the following loop is executed on the eight-word direct mapped cache from Figure 8.7? Assume that the cache is initially empty.

```
      addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0x24($0)
      addi $t0, $t0, -1
      j    loop
done:
```

Solution: Memory addresses 0x4 and 0x24 both map to set 1. During the initial execution of the loop, data at address 0x4 is loaded into set 1 of the cache. Then data at address 0x24 is loaded into set 1, evicting the data from address 0x4. Upon the second execution of the loop, the pattern repeats and the cache must refetch data at address 0x4, evicting data from address 0x24. The two addresses conflict, and the miss rate is 100%.

Multi-way Set Associative Cache

An *N-way set associative* cache reduces conflicts by providing *N* blocks in each set where data mapping to that set might be found. Each memory address still maps to a specific set, but it can map to any one of the *N* blocks in the set. Hence, a direct mapped cache is another name for a one-way set associative cache. *N* is also called the *degree of associativity* of the cache.

Figure 8.9 shows the hardware for a $C = 8$-word, $N = 2$-way set associative cache. The cache now has only $S = 4$ sets rather than 8. Thus, only

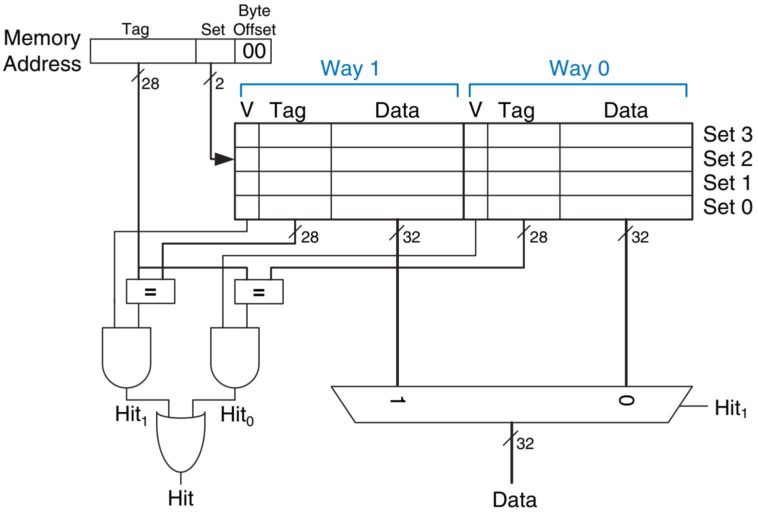


Figure 8.9 **Two-way set associative cache**

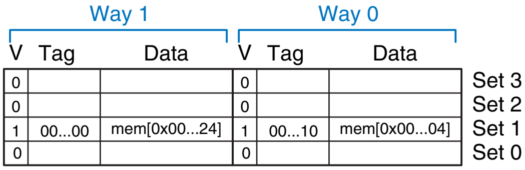
$\log_2 4 = 2$ set bits rather than 3 are used to select the set. The tag increases from 27 to 28 bits. Each set contains two *ways* or degrees of associativity. Each way consists of a data block and the valid and tag bits. The cache reads blocks from both ways in the selected set and checks the tags and valid bits for a hit. If a hit occurs in one of the ways, a multiplexer selects data from that way.

Set associative caches generally have lower miss rates than direct mapped caches of the same capacity, because they have fewer conflicts. However, set associative caches are usually slower and somewhat more expensive to build because of the output multiplexer and additional comparators. They also raise the question of which way to replace when both ways are full; this is addressed further in Section 8.3.3. Most commercial systems use set associative caches.

Example 8.8 SET ASSOCIATIVE CACHE MISS RATE

Repeat Example 8.7 using the eight-word two-way set associative cache from Figure 8.9.

Solution: Both memory accesses, to addresses 0x4 and 0x24, map to set 1. However, the cache has two ways, so it can accommodate data from both addresses. During the first loop iteration, the empty cache misses both addresses and loads both words of data into the two ways of set 1, as shown in Figure 8.10. On the next four iterations, the cache hits. Hence, the miss rate is 2/10 = 20%. Recall that the direct mapped cache of the same size from Example 8.7 had a miss rate of 100%.



Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...00	mem[0x00...24]	1	00...10	mem[0x00...04]	Set 1
0			0			Set 0

Figure 8.10 Two-way set associative cache contents

Fully Associative Cache

A *fully associative* cache contains a single set with B ways, where B is the number of blocks. A memory address can map to a block in any of these ways. A fully associative cache is another name for a B-way set associative cache with one set.

Figure 8.11 shows the SRAM array of a fully associative cache with eight blocks. Upon a data request, eight tag comparisons (not shown) must be made, because the data could be in any block. Similarly, an 8:1 multiplexer chooses the proper data if a hit occurs. Fully associative

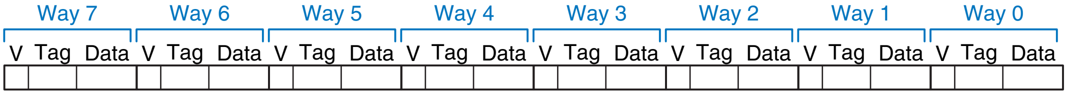


Figure 8.11 Eight-block fully associative cache

caches tend to have the fewest conflict misses for a given cache capacity, but they require more hardware for additional tag comparisons. They are best suited to relatively small caches because of the large number of comparators.

Block Size

The previous examples were able to take advantage only of temporal locality, because the block size was one word. To exploit spatial locality, a cache uses larger blocks to hold several consecutive words.

The advantage of a block size greater than one is that when a miss occurs and the word is fetched into the cache, the adjacent words in the block are also fetched. Therefore, subsequent accesses are more likely to hit because of spatial locality. However, a large block size means that a fixed-size cache will have fewer blocks. This may lead to more conflicts, increasing the miss rate. Moreover, it takes more time to fetch the missing cache block after a miss, because more than one data word is fetched from main memory. The time required to load the missing block into the cache is called the *miss penalty*. If the adjacent words in the block are not accessed later, the effort of fetching them is wasted. Nevertheless, most real programs benefit from larger block sizes.

Figure 8.12 shows the hardware for a $C = 8$-word direct mapped cache with a $b = 4$-word block size. The cache now has only $B = C/b = 2$ blocks. A direct mapped cache has one block in each set, so this cache

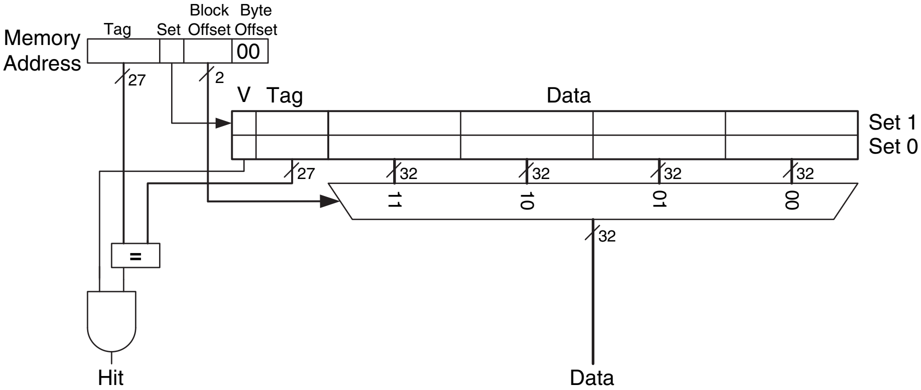


Figure 8.12 Direct mapped cache with two sets and a four-word block size

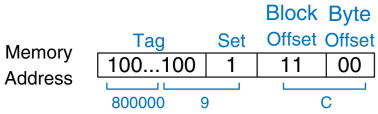

Memory
Address
Tag
Set
Block
Offset
Byte
Offset
100...100
1
11
00
800000
9
C

Figure 8.13 Cache fields for address 0x8000009C when mapping to the cache of Figure 8.12

is organized as two sets. Thus, only $\log_2 2 = 1$ bit is used to select the set. A multiplexer is now needed to select the word within the block. The multiplexer is controlled by the $\log_2 4 = 2$ *block offset bits* of the address. The most significant 27 address bits form the tag. Only one tag is needed for the entire block, because the words in the block are at consecutive addresses.

Figure 8.13 shows the cache fields for address 0x8000009C when it maps to the direct mapped cache of Figure 8.12. The byte offset bits are always 0 for word accesses. The next $\log_2 b = 2$ block offset bits indicate the word within the block. And the next bit indicates the set. The remaining 27 bits are the tag. Therefore, word 0x8000009C maps to set 1, word 3 in the cache. The principle of using larger block sizes to exploit spatial locality also applies to associative caches.

Example 8.9 SPATIAL LOCALITY WITH A DIRECT MAPPED CACHE

Repeat Example 8.6 for the eight-word direct mapped cache with a four-word block size.

Solution: Figure 8.14 shows the contents of the cache after the first memory access. On the first loop iteration, the cache misses on the access to memory address 0x4. This access loads data at addresses 0x0 through 0xC into the cache block. All subsequent accesses (as shown for address 0xC) hit in the cache. Hence, the miss rate is 1/15 = 6.67%.

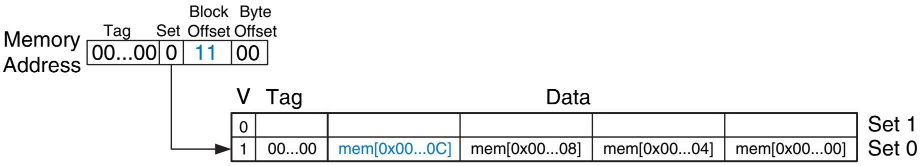

Memory
Address
Tag
Set
Block
Offset
Byte
Offset
00...00
0
11
00
V
Tag
Data
0
1
00...00
mem[0x00...0C]
mem[0x00...08]
mem[0x00...04]
mem[0x00...00]
Set 1
Set 0

Figure 8.14 Cache contents with a block size *b* of four words

Putting it All Together

Caches are organized as two-dimensional arrays. The rows are called sets, and the columns are called ways. Each entry in the array

Table 8.2 **Cache organizations**

Organization	Number of Ways (N)	Number of Sets (S)
Direct Mapped	1	B
Set Associative	$1 < N < B$	B/N
Fully Associative	B	1

consists of a data block and its associated valid and tag bits. Caches are characterized by

- capacity C
- block size b (and number of blocks, $B = C/b$)
- number of blocks in a set (N)

Table 8.2 summarizes the various cache organizations. Each address in memory maps to only one set but can be stored in any of the ways.

Cache capacity, associativity, set size, and block size are typically powers of 2. This makes the cache fields (tag, set, and block offset bits) subsets of the address bits.

Increasing the associativity N usually reduces the miss rate caused by conflicts. But higher associativity requires more tag comparators. Increasing the block size b takes advantage of spatial locality to reduce the miss rate. However, it decreases the number of sets in a fixed sized cache and therefore could lead to more conflicts. It also increases the miss penalty.

8.3.3 What Data is Replaced?

In a direct mapped cache, each address maps to a unique block and set. If a set is full when new data must be loaded, the block in that set is replaced with the new data. In set associative and fully associative caches, the cache must choose which block to evict when a cache set is full. The principle of temporal locality suggests that the best choice is to evict the least recently used block, because it is least likely to be used again soon. Hence, most associative caches have a *least recently used* (*LRU*) replacement policy.

In a two-way set associative cache, a *use bit*, U, indicates which way within a set was least recently used. Each time one of the ways is used, U is adjusted to indicate the other way. For set associative caches with more than two ways, tracking the least recently used way becomes complicated. To simplify the problem, the ways are often divided into two groups and U indicates which *group* of ways was least recently used. Upon replacement, the new block replaces a random block within the least recently used group. Such a policy is called *pseudo-LRU* and is good enough in practice.

Example 8.10 LRU REPLACEMENT

Show the contents of an eight-word two-way set associative cache after executing the following code. Assume LRU replacement, a block size of one word, and an initially empty cache.

```
lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

Solution: The first two instructions load data from memory addresses 0x4 and 0x24 into set 1 of the cache, shown in Figure 8.15(a). $U = 0$ indicates that data in way 0 was the least recently used. The next memory access, to address 0x54, also maps to set 1 and replaces the least recently used data in way 0, as shown in Figure 8.15(b). The use bit U is set to 1 to indicate that data in way 1 was the least recently used.

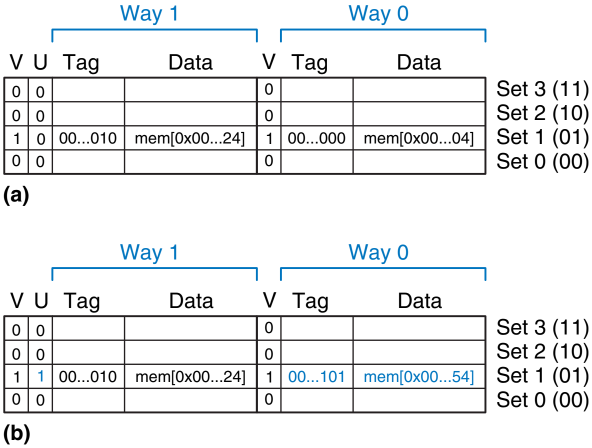

Way 1
Way 0
V U Tag Data V Tag Data
0 0 0
0 0 0
1 0 00...010 mem[0x00...24] 1 00...000 mem[0x00...04]
0 0 0
Set 3 (11)
Set 2 (10)
Set 1 (01)
Set 0 (00)
(a)
Way 1
Way 0
V U Tag Data V Tag Data
0 0 0
0 0 0
1 1 00...010 mem[0x00...24] 1 00...101 mem[0x00...54]
0 0 0
Set 3 (11)
Set 2 (10)
Set 1 (01)
Set 0 (00)
(b)

Figure 8.15 Two-way associative cache with LRU replacement

8.3.4 Advanced Cache Design*

Modern systems use multiple levels of caches to decrease memory access time. This section explores the performance of a two-level caching system and examines how block size, associativity, and cache capacity affect miss rate. The section also describes how caches handle stores, or writes, by using a write-through or write-back policy.

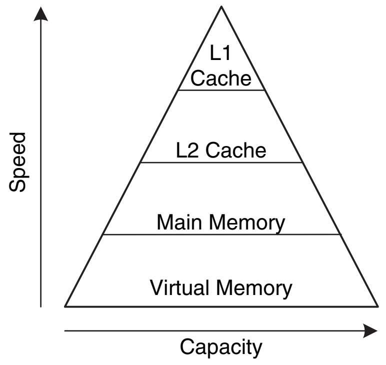


Figure 8.16 **Memory hierarchy with two levels of cache**

Multiple-Level Caches

Large caches are beneficial because they are more likely to hold data of interest and therefore have lower miss rates. However, large caches tend to be slower than small ones. Modern systems often use at least two levels of caches, as shown in Figure 8.16. The first-level (L1) cache is small enough to provide a one- or two-cycle access time. The second-level (L2) cache is also built from SRAM but is larger, and therefore slower, than the L1 cache. The processor first looks for the data in the L1 cache. If the L1 cache misses, the processor looks in the L2 cache. If the L2 cache misses, the processor fetches the data from main memory. Many modern systems add even more levels of cache to the memory hierarchy, because accessing main memory is so slow.

Example 8.11 SYSTEM WITH AN L2 CACHE

Use the system of Figure 8.16 with access times of 1, 10, and 100 cycles for the L1 cache, L2 cache, and main memory, respectively. Assume that the L1 and L2 caches have miss rates of 5% and 20%, respectively. Specifically, of the 5% of accesses that miss the L1 cache, 20% of those also miss the L2 cache. What is the average memory access time (*AMAT*)?

Solution: Each memory access checks the L1 cache. When the L1 cache misses (5% of the time), the processor checks the L2 cache. When the L2 cache misses (20% of the time), the processor fetches the data from main memory. Using Equation 8.2, we calculate the average memory access time as follows: 1 cycle + 0.05[10 cycles + 0.2(100 cycles)] = 2.5 cycles

The L2 miss rate is high because it receives only the "hard" memory accesses, those that miss in the L1 cache. If all accesses went directly to the L2 cache, the L2 miss rate would be about 1%.

Reducing Miss Rate

Cache misses can be reduced by changing capacity, block size, and/or associativity. The first step to reducing the miss rate is to understand the causes of the misses. The misses can be classified as compulsory, capacity, and conflict. The first request to a cache block is called a *compulsory miss*, because the block must be read from memory regardless of the cache design. *Capacity misses* occur when the cache is too small to hold all concurrently used data. *Conflict misses* are caused when several addresses map to the same set and evict blocks that are still needed.

Changing cache parameters can affect one or more type of cache miss. For example, increasing cache capacity can reduce conflict and capacity misses, but it does not affect compulsory misses. On the other hand, increasing block size could reduce compulsory misses (due to spatial locality) but might actually *increase* conflict misses (because more addresses would map to the same set and could conflict).

Memory systems are complicated enough that the best way to evaluate their performance is by running benchmarks while varying cache parameters. Figure 8.17 plots miss rate versus cache size and degree of associativity for the SPEC2000 benchmark. This benchmark has a small number of compulsory misses, shown by the dark region near the x-axis. As expected, when cache size increases, capacity misses decrease. Increased associativity, especially for small caches, decreases the number of conflict misses shown along the top of the curve. Increasing associativity beyond four or eight ways provides only small decreases in miss rate.

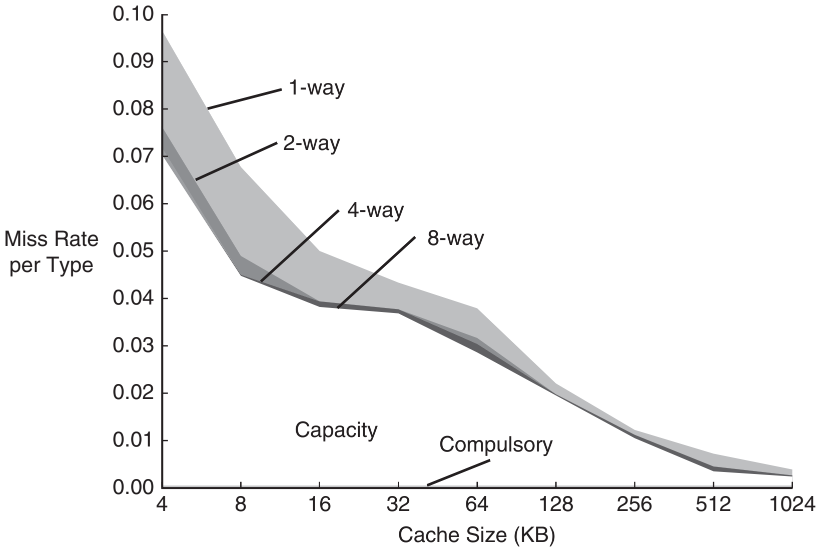


Figure 8.17 Miss rate versus cache size and associativity on SPEC2000 benchmark Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012.

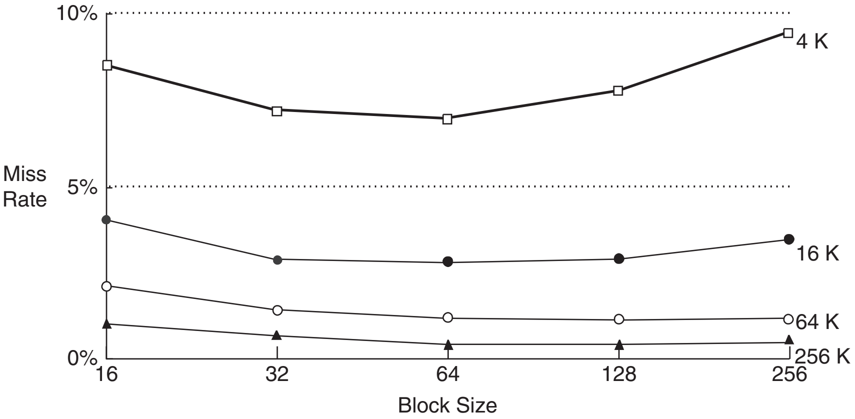


Figure 8.18 **Miss rate versus block size and cache size on SPEC92 benchmark** Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012.

As mentioned, miss rate can also be decreased by using larger block sizes that take advantage of spatial locality. But as block size increases, the number of sets in a fixed-size cache decreases, increasing the probability of conflicts. Figure 8.18 plots miss rate versus block size (in number of bytes) for caches of varying capacity. For small caches, such as the 4-KB cache, increasing the block size beyond 64 bytes *increases* the miss rate because of conflicts. For larger caches, increasing the block size beyond 64 bytes does not change the miss rate. However, large block sizes might still increase execution time because of the larger miss penalty, the time required to fetch the missing cache block from main memory.

Write Policy

The previous sections focused on memory loads. Memory stores, or writes, follow a similar procedure as loads. Upon a memory store, the processor checks the cache. If the cache misses, the cache block is fetched from main memory into the cache, and then the appropriate word in the cache block is written. If the cache hits, the word is simply written to the cache block.

Caches are classified as either write-through or write-back. In a *write-through* cache, the data written to a cache block is simultaneously written to main memory. In a *write-back* cache, a *dirty bit* (*D*) is associated with each cache block. *D* is 1 when the cache block has been written and 0 otherwise. Dirty cache blocks are written back to main memory only when they are evicted from the cache. A write-through cache requires no dirty bit but usually requires more main memory writes than a write-back cache. Modern caches are usually write-back, because main memory access time is so large.

Example 8.12 WRITE-THROUGH VERSUS WRITE-BACK

Suppose a cache has a block size of four words. How many main memory accesses are required by the following code when using each write policy: write-through or write-back?

```
sw $t0, 0x0($0)
sw $t0, 0xC($0)
sw $t0, 0x8($0)
sw $t0, 0x4($0)
```

Solution: All four store instructions write to the same cache block. With a write-through cache, each store instruction writes a word to main memory, requiring four main memory writes. A write-back policy requires only one main memory access, when the dirty cache block is evicted.

8.3.5 The Evolution of MIPS Caches*

Table 8.3 traces the evolution of cache organizations used by the MIPS processor from 1985 to 2010. The major trends are the introduction of multiple levels of cache, larger cache capacity, and increased associativity. These trends are driven by the growing disparity between CPU frequency and main memory speed and the decreasing cost of transistors. The growing difference between CPU and memory speeds necessitates a lower miss rate to avoid the main memory bottleneck, and the decreasing cost of transistors allows larger cache sizes.

Table 8.3 **MIPS cache evolution***

Year	CPU	MHz	L1 Cache	L2 Cache
1985	R2000	16.7	none	none
1990	R3000	33	32 KB direct mapped	none
1991	R4000	100	8 KB direct mapped	1 MB direct mapped
1995	R10000	250	32 KB two-way	4 MB two-way
2001	R14000	600	32 KB two-way	16 MB two-way
2004	R16000A	800	64 KB two-way	16 MB two-way
2010	MIPS32 1074K	1500	32 KB	variable size

[*] Adapted from D. Sweetman, *See MIPS Run*, Morgan Kaufmann, 1999.

8.4 VIRTUAL MEMORY

Most modern computer systems use a *hard drive* made of magnetic or solid state storage as the lowest level in the memory hierarchy (see Figure 8.4). Compared with the ideal large, fast, cheap memory, a hard drive is large and cheap but terribly slow. It provides a much larger capacity than is possible with a cost-effective main memory (DRAM). However, if a significant fraction of memory accesses involve the hard drive, performance is dismal. You may have encountered this on a PC when running too many programs at once.

Figure 8.19 shows a hard drive made of magnetic storage, also called a *hard disk*, with the lid of its case removed. As the name implies, the hard disk contains one or more rigid disks or *platters*, each of which has a *read/write head* on the end of a long triangular arm. The head moves to the correct location on the disk and reads or writes data magnetically as the disk rotates beneath it. The head takes several milliseconds to *seek* the correct location on the disk, which is fast from a human perspective but millions of times slower than the processor.

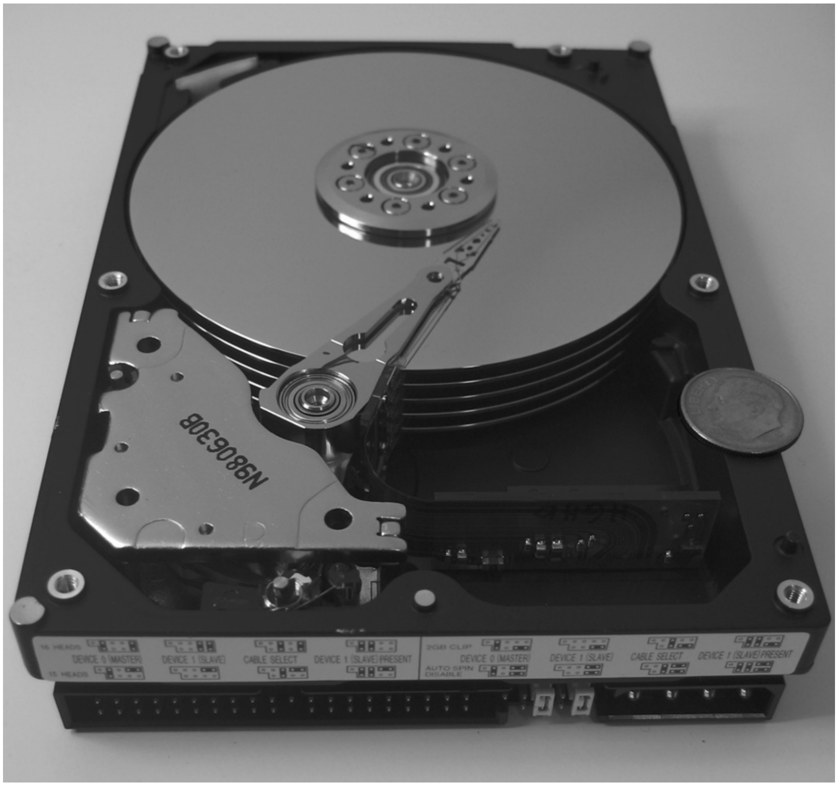

N9806308
16 HEADS
15 HEADS
DEVICE 0 (MASTER)
DEVICE 1 (SLAVE)
CABLE SELECT
DEVICE 1 (SLAVE) PRESENT
2GB CLIP
AUTO SPIN DISABLE
DEVICE 0 (MASTER)
DEVICE 1 (SLAVE)
CABLE SELECT
DEVICE 1 (SLAVE) PRESENT

Figure 8.19 Hard disk

The objective of adding a hard drive to the memory hierarchy is to inexpensively give the illusion of a very large memory while still providing the speed of faster memory for most accesses. A computer with only 128 MB of DRAM, for example, could effectively provide 2 GB of memory using the hard drive. This larger 2-GB memory is called *virtual memory*, and the smaller 128-MB main memory is called *physical memory*. We will use the term physical memory to refer to main memory throughout this section.

A computer with 32-bit addresses can access a maximum of 2^{32} bytes = 4 GB of memory. This is one of the motivations for moving to 64-bit computers, which can access far more memory.

Programs can access data anywhere in virtual memory, so they must use *virtual addresses* that specify the location in virtual memory. The physical memory holds a subset of most recently accessed virtual memory. In this way, physical memory acts as a cache for virtual memory. Thus, most accesses hit in physical memory at the speed of DRAM, yet the program enjoys the capacity of the larger virtual memory.

Virtual memory systems use different terminologies for the same caching principles discussed in Section 8.3. Table 8.4 summarizes the analogous terms. Virtual memory is divided into *virtual pages*, typically 4 KB in size. Physical memory is likewise divided into *physical pages* of the same size. A virtual page may be located in physical memory (DRAM) or on the hard drive. For example, Figure 8.20 shows a virtual memory that is larger than physical memory. The rectangles indicate pages. Some virtual pages are present in physical memory, and some are located on the hard drive. The process of determining the physical address from the virtual address is called *address translation*. If the processor attempts to access a virtual address that is not in physical memory, a *page fault* occurs, and the operating system loads the page from the hard drive into physical memory.

To avoid page faults caused by conflicts, any virtual page can map to any physical page. In other words, physical memory behaves as a fully associative cache for virtual memory. In a conventional fully associative cache, every cache block has a comparator that checks the most significant address bits against a tag to determine whether the request hits in

Table 8.4 Analogous cache and virtual memory terms

Cache	Virtual Memory
Block	Page
Block size	Page size
Block offset	Page offset
Miss	Page fault
Tag	Virtual page number

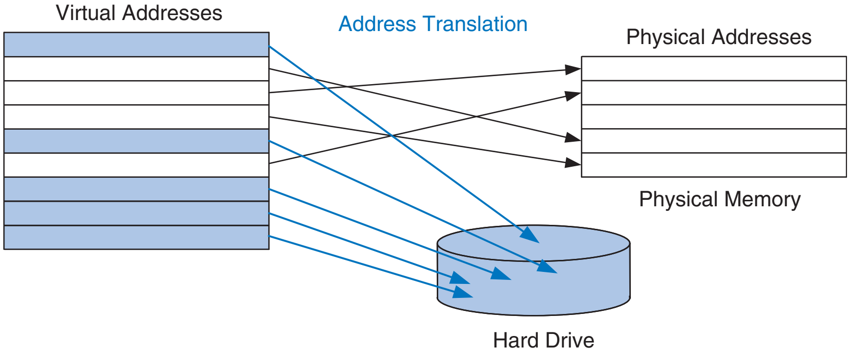


Figure 8.20 **Virtual and physical pages**

the block. In an analogous virtual memory system, each physical page would need a comparator to check the most significant virtual address bits against a tag to determine whether the virtual page maps to that physical page.

A realistic virtual memory system has so many physical pages that providing a comparator for each page would be excessively expensive. Instead, the virtual memory system uses a page table to perform address translation. A page table contains an entry for each virtual page, indicating its location in physical memory or that it is on the hard drive. Each load or store instruction requires a page table access followed by a physical memory access. The page table access translates the virtual address used by the program to a physical address. The physical address is then used to actually read or write the data.

The page table is usually so large that it is located in physical memory. Hence, each load or store involves two physical memory accesses: a page table access, and a data access. To speed up address translation, a translation lookaside buffer (TLB) caches the most commonly used page table entries.

The remainder of this section elaborates on address translation, page tables, and TLBs.

8.4.1 Address Translation

In a system with virtual memory, programs use virtual addresses so that they can access a large memory. The computer must translate these virtual addresses to either find the address in physical memory or take a page fault and fetch the data from the hard drive.

Recall that virtual memory and physical memory are divided into pages. The most significant bits of the virtual or physical address specify the virtual or physical *page number*. The least significant bits specify the word within the page and are called the *page offset*.

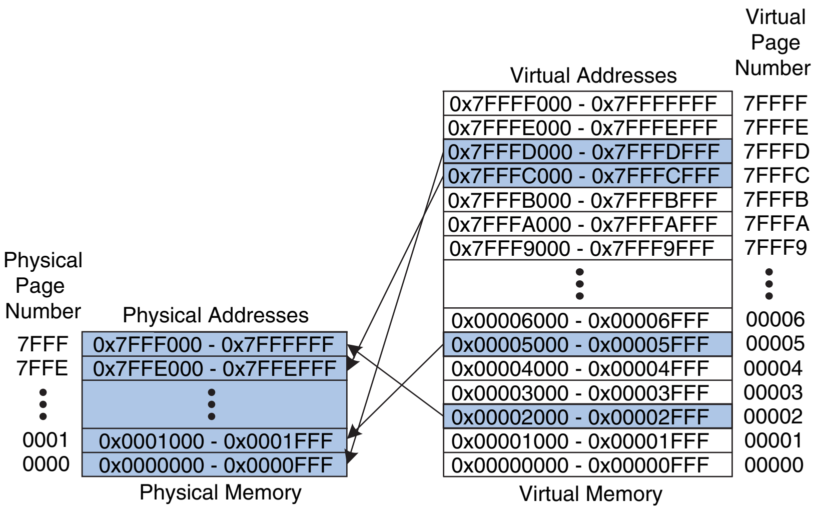

Virtual Page Number
Virtual Addresses
0x7FFFF000 - 0x7FFFFFFF
0x7FFFE000 - 0x7FFFEFFF
0x7FFFD000 - 0x7FFFDFFF
0x7FFFC000 - 0x7FFFCFFF
0x7FFFB000 - 0x7FFFBFFF
0x7FFFA000 - 0x7FFFAFFF
0x7FFF9000 - 0x7FFF9FFF
0x00006000 - 0x00006FFF
0x00005000 - 0x00005FFF
0x00004000 - 0x00004FFF
0x00003000 - 0x00003FFF
0x00002000 - 0x00002FFF
0x00001000 - 0x00001FFF
0x00000000 - 0x00000FFF
7FFFF
7FFFE
7FFFD
7FFFC
7FFFB
7FFFA
7FFF9
00006
00005
00004
00003
00002
00001
00000
Virtual Memory
Physical Page Number
Physical Addresses
7FFF
7FFE
0001
0000
0x7FFF000 - 0x7FFFFFF
0x7FFE000 - 0x7FFEFFF
0x0001000 - 0x0001FFF
0x0000000 - 0x0000FFF
Physical Memory

Figure 8.21 **Physical and virtual pages**

Figure 8.21 illustrates the page organization of a virtual memory system with 2 GB of virtual memory and 128 MB of physical memory divided into 4-KB pages. MIPS accommodates 32-bit addresses. With a 2-GB = 2^{31}-byte virtual memory, only the least significant 31 virtual address bits are used; the 32nd bit is always 0. Similarly, with a 128-MB = 2^{27}-byte physical memory, only the least significant 27 physical address bits are used; the upper 5 bits are always 0.

Because the page size is 4 KB = 2^{12} bytes, there are $2^{31}/2^{12} = 2^{19}$ virtual pages and $2^{27}/2^{12} = 2^{15}$ physical pages. Thus, the virtual and physical page numbers are 19 and 15 bits, respectively. Physical memory can only hold up to 1/16th of the virtual pages at any given time. The rest of the virtual pages are kept on the hard drive.

Figure 8.21 shows virtual page 5 mapping to physical page 1, virtual page 0x7FFFC mapping to physical page 0x7FFE, and so forth. For example, virtual address 0x53F8 (an offset of 0x3F8 within virtual page 5) maps to physical address 0x13F8 (an offset of 0x3F8 within physical page 1). The least significant 12 bits of the virtual and physical addresses are the same (0x3F8) and specify the page offset within the virtual and physical pages. Only the page number needs to be translated to obtain the physical address from the virtual address.

Figure 8.22 illustrates the translation of a virtual address to a physical address. The least significant 12 bits indicate the page offset and require no translation. The upper 19 bits of the virtual address specify the *virtual page number* (*VPN*) and are translated to a 15-bit *physical page number* (*PPN*). The next two sections describe how page tables and TLBs are used to perform this address translation.

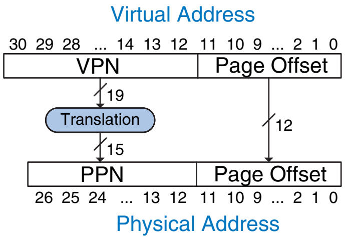


Figure 8.22 **Translation from virtual address to physical address**

Example 8.13 VIRTUAL ADDRESS TO PHYSICAL ADDRESS TRANSLATION

Find the physical address of virtual address 0x247C using the virtual memory system shown in Figure 8.21.

Solution: The 12-bit page offset (0x47C) requires no translation. The remaining 19 bits of the virtual address give the virtual page number, so virtual address 0x247C is found in virtual page 0x2. In Figure 8.21, virtual page 0x2 maps to physical page 0x7FFF. Thus, virtual address 0x247C maps to physical address 0x7FFF47C.

V	Physical Page Number	Virtual Page Number
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Page Table

Figure 8.23 **The page table for Figure 8.21**

8.4.2 The Page Table

The processor uses a *page table* to translate virtual addresses to physical addresses. The page table contains an entry for each virtual page. This entry contains a physical page number and a valid bit. If the valid bit is 1, the virtual page maps to the physical page specified in the entry. Otherwise, the virtual page is found on the hard drive.

Because the page table is so large, it is stored in physical memory. Let us assume for now that it is stored as a contiguous array, as shown in Figure 8.23. This page table contains the mapping of the memory system of Figure 8.21. The page table is indexed with the virtual page number (VPN). For example, entry 5 specifies that virtual page 5 maps to physical page 1. Entry 6 is invalid ($V = 0$), so virtual page 6 is located on the hard drive.

Example 8.14 USING THE PAGE TABLE TO PERFORM ADDRESS TRANSLATION

Find the physical address of virtual address 0x247C using the page table shown in Figure 8.23.

Solution: Figure 8.24 shows the virtual address to physical address translation for virtual address 0x247C. The 12-bit page offset requires no translation. The remaining 19 bits of the virtual address are the virtual page number, 0x2, and give the index into the page table. The page table maps virtual page 0x2 to physical page 0x7FFF. So, virtual address 0x247C maps to physical address 0x7FFF47C. The least significant 12 bits are the same in both the physical and the virtual address.

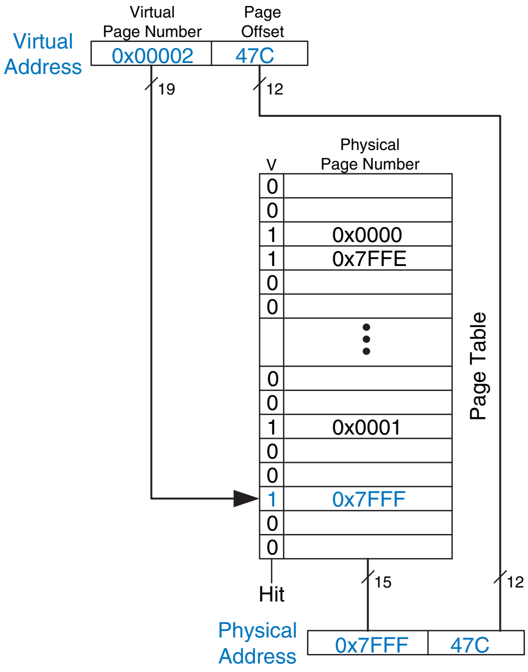

Virtual Address
Virtual Page Number
Page Offset
0x00002
47C
19
12
V
Physical Page Number
0
0
1 0x0000
1 0x7FFE
0
0
0
0
1 0x0001
0
0
1 0x7FFF
0
0
Page Table
Hit
15
12
Physical Address
0x7FFF
47C

Figure 8.24 **Address translation using the page table**

The page table can be stored anywhere in physical memory, at the discretion of the OS. The processor typically uses a dedicated register, called the *page table register*, to store the base address of the page table in physical memory.

To perform a load or store, the processor must first translate the virtual address to a physical address and then access the data at that physical address. The processor extracts the virtual page number from the virtual address and adds it to the page table register to find the physical address of the page table entry. The processor then reads this page table entry from physical memory to obtain the physical page number. If the entry is valid, it merges this physical page number with the page offset to create the physical address. Finally, it reads or writes data at this physical address. Because the page table is stored in physical memory, each load or store involves two physical memory accesses.

8.4.3 The Translation Lookaside Buffer

Virtual memory would have a severe performance impact if it required a page table read on every load or store, doubling the delay of loads and stores. Fortunately, page table accesses have great temporal locality. The temporal and spatial locality of data accesses and the large page size mean that many consecutive loads or stores are likely to reference the same page. Therefore, if the processor remembers the last page table entry that it read, it can probably reuse this translation without rereading the page table. In general, the processor can keep the last several page table entries in a small cache called a *translation lookaside buffer* (*TLB*). The processor "looks aside" to find the translation in the TLB before having to access the page table in physical memory. In real programs, the vast majority of accesses hit in the TLB, avoiding the time-consuming page table reads from physical memory.

A TLB is organized as a fully associative cache and typically holds 16 to 512 entries. Each TLB entry holds a virtual page number and its corresponding physical page number. The TLB is accessed using the virtual page number. If the TLB hits, it returns the corresponding physical page number. Otherwise, the processor must read the page table in physical memory. The TLB is designed to be small enough that it can be accessed in less than one cycle. Even so, TLBs typically have a hit rate of greater than 99%. The TLB decreases the number of memory accesses required for most load or store instructions from two to one.

Example 8.15 USING THE TLB TO PERFORM ADDRESS TRANSLATION

Consider the virtual memory system of Figure 8.21. Use a two-entry TLB or explain why a page table access is necessary to translate virtual addresses 0x247C and 0x5FB0 to physical addresses. Suppose the TLB currently holds valid translations of virtual pages 0x2 and 0x7FFFD.

Solution: Figure 8.25 shows the two-entry TLB with the request for virtual address 0x247C. The TLB receives the virtual page number of the incoming address, 0x2, and compares it to the virtual page number of each entry. Entry 0 matches and is valid, so the request hits. The translated physical address is the physical page number of the matching entry, 0x7FFF, concatenated with the page offset of the virtual address. As always, the page offset requires no translation.

The request for virtual address 0x5FB0 misses in the TLB. So, the request is forwarded to the page table for translation.

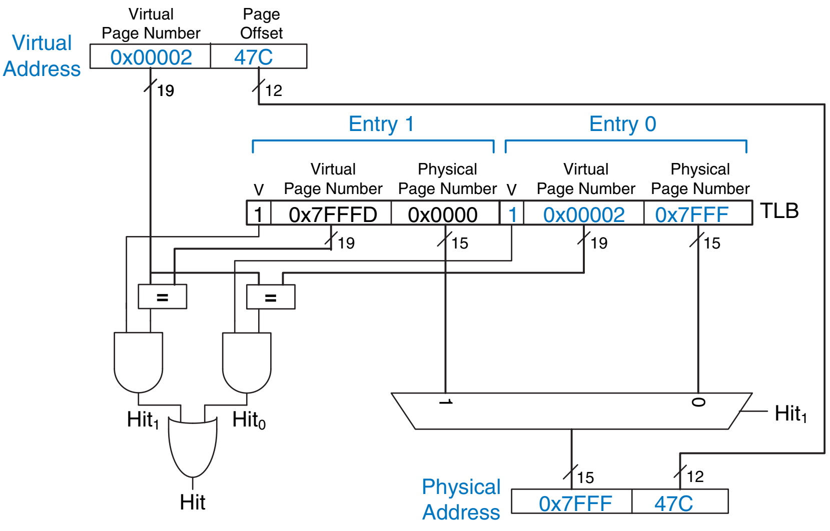

Virtual Address
Virtual Page Number
Page Offset
0x00002
47C
19
12
Entry 1
Entry 0
V
Virtual Page Number
Physical Page Number
V
Virtual Page Number
Physical Page Number
1
0x7FFFD
0x0000
1
0x00002
0x7FFF
TLB
19
15
19
15
=
=
1
0
Hit$_1$
Hit$_1$
Hit$_0$
Hit
Physical Address
15
12
0x7FFF
47C

Figure 8.25 **Address translation using a two-entry TLB**

8.4.4 Memory Protection

So far, this section has focused on using virtual memory to provide a fast, inexpensive, large memory. An equally important reason to use virtual memory is to provide protection between concurrently running programs.

As you probably know, modern computers typically run several programs or *processes* at the same time. All of the programs are simultaneously present in physical memory. In a well-designed computer system, the programs should be protected from each other so that no program can crash or hijack another program. Specifically, no program should be able to access another program's memory without permission. This is called *memory protection*.

Virtual memory systems provide memory protection by giving each program its own *virtual address space*. Each program can use as much memory as it wants in that virtual address space, but only a portion of the virtual address space is in physical memory at any given time. Each program can use its entire virtual address space without having to worry about where other programs are physically located. However, a program can access only those physical pages that are mapped in its page table. In this way, a program cannot accidentally or maliciously access another program's physical pages, because they are not mapped in its page table. In some cases, multiple programs access common instructions or data. The operating system adds control bits to each page table entry to determine which programs, if any, can write to the shared physical pages.

8.4.5 Replacement Policies*

Virtual memory systems use write-back and an approximate least recently used (LRU) replacement policy. A write-through policy, where each write to physical memory initiates a write to the hard drive, would be impractical. Store instructions would operate at the speed of the hard drive instead of the speed of the processor (milliseconds instead of nanoseconds). Under the writeback policy, the physical page is written back to the hard drive only when it is evicted from physical memory. Writing the physical page back to the hard drive and reloading it with a different virtual page is called *paging*, and the hard drive in a virtual memory system is sometimes called *swap space*. The processor pages out one of the least recently used physical pages when a page fault occurs, then replaces that page with the missing virtual page. To support these replacement policies, each page table entry contains two additional status bits: a dirty bit D and a use bit U.

The dirty bit is 1 if any store instructions have changed the physical page since it was read from the hard drive. When a physical page is paged out, it needs to be written back to the hard drive only if its dirty bit is 1; otherwise, the hard drive already holds an exact copy of the page.

The use bit is 1 if the physical page has been accessed recently. As in a cache system, exact LRU replacement would be impractically complicated. Instead, the OS approximates LRU replacement by periodically resetting all the use bits in the page table. When a page is accessed, its use bit is set to 1. Upon a page fault, the OS finds a page with $U = 0$ to page out of physical memory. Thus, it does not necessarily replace the least recently used page, just one of the least recently used pages.

8.4.6 Multilevel Page Tables*

Page tables can occupy a large amount of physical memory. For example, the page table from the previous sections for a 2 GB virtual memory with 4 KB pages would need 2^{19} entries. If each entry is 4 bytes, the page table is $2^{19} \times 2^2$ bytes $= 2^{21}$ bytes $= 2$ MB.

To conserve physical memory, page tables can be broken up into multiple (usually two) levels. The first-level page table is always kept in physical memory. It indicates where small second-level page tables are stored in virtual memory. The second-level page tables each contain the actual translations for a range of virtual pages. If a particular range of translations is not actively used, the corresponding second-level page table can be paged out to the hard drive so it does not waste physical memory.

In a two-level page table, the virtual page number is split into two parts: the *page table number* and the *page table offset*, as shown in Figure 8.26. The page table number indexes the first-level page table, which must reside in physical memory. The first-level page table entry gives the base address of the second-level page table or indicates that it must be fetched from the

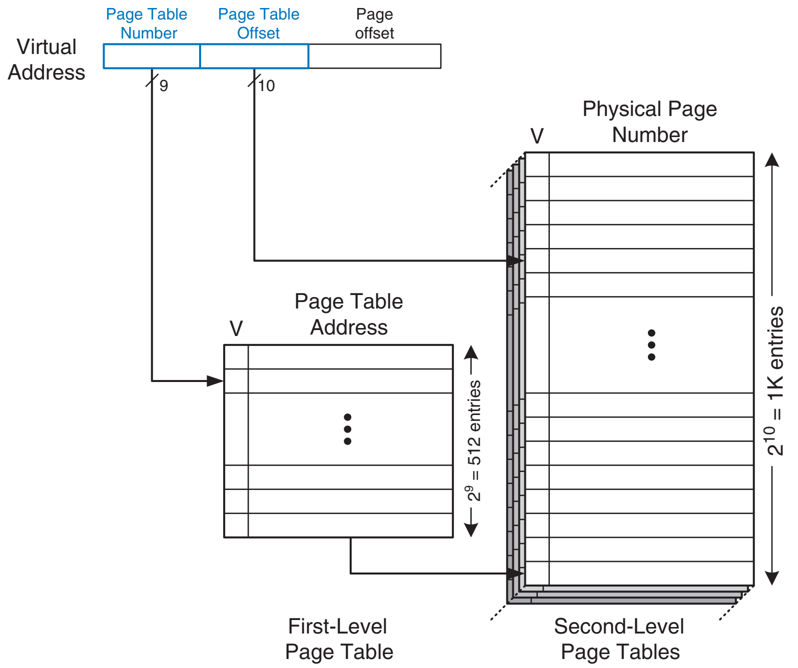

Virtual Address
Page Table Number
Page Table Offset
Page offset
9
10
V
Physical Page Number
V
Page Table Address
2^9 = 512 entries
2^{10} = 1K entries
First-Level Page Table
Second-Level Page Tables

Figure 8.26 **Hierarchical page tables**

hard drive when *V* is 0. The page table offset indexes the second-level page table. The remaining 12 bits of the virtual address are the page offset, as before, for a page size of $2^{12} = 4$ KB.

In Figure 8.26 the 19-bit virtual page number is broken into 9 and 10 bits, to indicate the page table number and the page table offset, respectively. Thus, the first-level page table has $2^9 = 512$ entries. Each of these 512 second-level page tables has $2^{10} = 1$ K entries. If each of the first- and second-level page table entries is 32 bits (4 bytes) and only two second-level page tables are present in physical memory at once, the hierarchical page table uses only (512 × 4 bytes) + 2 × (1 K × 4 bytes) = 10 KB of physical memory. The two-level page table requires a fraction of the physical memory needed to store the entire page table (2 MB). The drawback of a two-level page table is that it adds yet another memory access for translation when the TLB misses.

Example 8.16 USING A MULTILEVEL PAGE TABLE FOR ADDRESS TRANSLATION

Figure 8.27 shows the possible contents of the two-level page table from Figure 8.26. The contents of only one second-level page table are shown. Using this two-level page table, describe what happens on an access to virtual address 0x003FEFB0.

Solution: As always, only the virtual page number requires translation. The most significant nine bits of the virtual address, 0x0, give the page table number, the index into the first-level page table. The first-level page table at entry 0x0 indicates that the second-level page table is resident in memory ($V = 1$) and its physical address is 0x2375000.

The next ten bits of the virtual address, 0x3FE, are the page table offset, which gives the index into the second-level page table. Entry 0 is at the bottom of the second-level page table, and entry 0x3FF is at the top. Entry 0x3FE in the second-level page table indicates that the virtual page is resident in physical memory ($V = 1$) and that the physical page number is 0x23F1. The physical page number is concatenated with the page offset to form the physical address, 0x23F1FB0.

Embedded systems are special-purpose computers that control some physical device. They typically consist of a microcontroller or digital signal processor connected to one or more hardware I/O devices. For example, a microwave oven may have a simple microcontroller to read the buttons, run the clock and timer, and turn the magnetron on and off at the appropriate times. Contrast embedded systems with *general-purpose computers*, such as PCs, which run multiple programs and typically interact more with the user than with a physical device. Systems such as smart phones blur the lines between embedded and general-purpose computing.

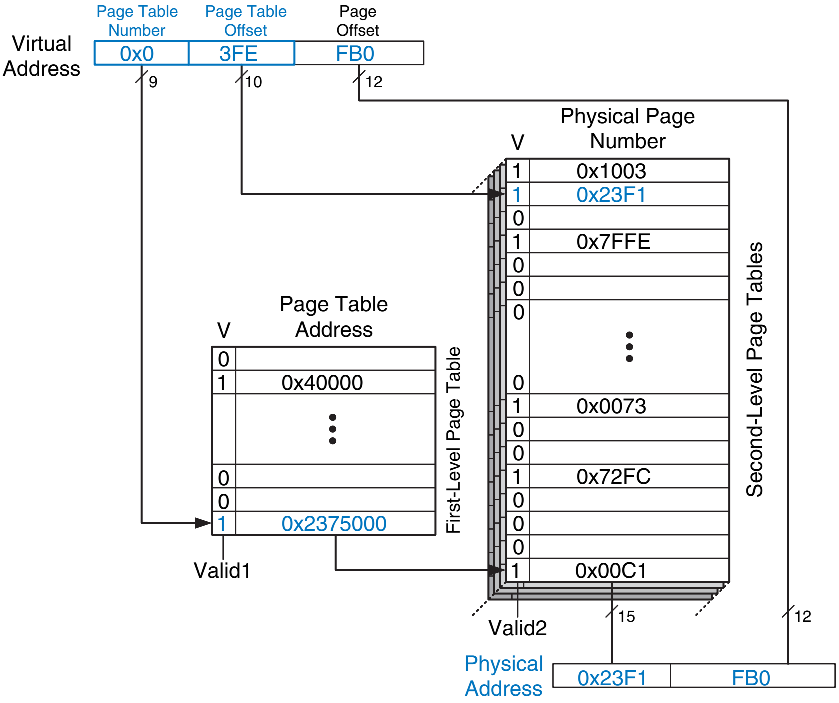

Virtual Address
Page Table Number
Page Table Offset
Page Offset
0x0
3FE
FB0
9
10
12
Physical Page Number
V
1 0x1003
1 0x23F1
0
1 0x7FFE
0
0
0
0
1 0x0073
0
0
1 0x72FC
0
0
0
1 0x00C1
Second-Level Page Tables
Page Table Address
V
0
1 0x40000
0
0
1 0x2375000
First-Level Page Table
Valid1
Valid2
15
12
Physical Address
0x23F1
FB0

Figure 8.27 Address translation using a two-level page table

8.5 I/O INTRODUCTION

Input/Output (I/O) *systems* are used to connect a computer with external devices called *peripherals*. In a personal computer, the devices typically include keyboards, monitors, printers, and wireless networks. In embedded

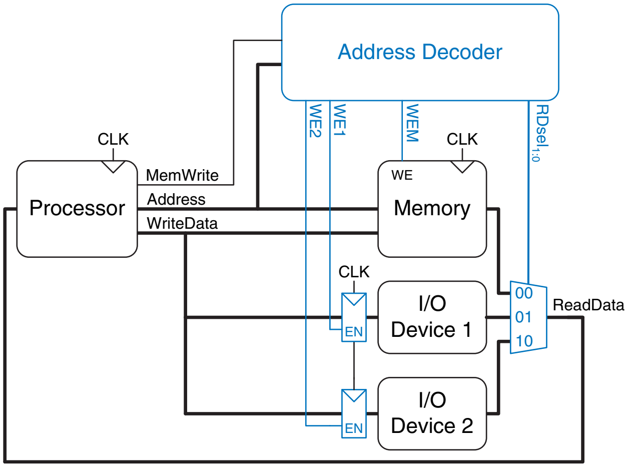


Figure 8.28 **Support hardware for memory-mapped I/O**

systems, devices could include a toaster's heating element, a doll's speech synthesizer, an engine's fuel injector, a satellite's solar panel positioning motors, and so forth. A processor accesses an I/O device using the address and data busses in the same way that it accesses memory.

A portion of the address space is dedicated to I/O devices rather than memory. For example, suppose that addresses in the range 0xFFFF0000 to 0xFFFFFFFF are used for I/O. Recall from Section 6.6.1 that these addresses are in a reserved portion of the memory map. Each I/O device is assigned one or more memory addresses in this range. A store to the specified address sends data to the device. A load receives data from the device. This method of communicating with I/O devices is called *memory-mapped I/O*.

In a system with memory-mapped I/O, a load or store may access either memory or an I/O device. Figure 8.28 shows the hardware needed to support two memory-mapped I/O devices. An *address decoder* determines which device communicates with the processor. It uses the *Address* and *MemWrite* signals to generate control signals for the rest of the hardware. The *ReadData* multiplexer selects between memory and the various I/O devices. Write-enabled registers hold the values written to the I/O devices.

Some architectures, notably x86, use specialized instructions instead of memory-mapped I/O to communicate with I/O devices. These instructions are of the following form, where `device1` and `device2` are the unique ID of the peripheral device:

```
lwio $t0, device1
swio $t0, device2
```

This type of communication with I/O devices is called *programmed I/O*.

Example 8.17 COMMUNICATING WITH I/O DEVICES

Suppose I/O Device 1 in Figure 8.28 is assigned the memory address 0xFFFFFFF4. Show the MIPS assembly code for writing the value 7 to I/O Device 1 and for reading the output value from I/O Device 1.

Solution: The following MIPS assembly code writes the value 7 to I/O Device 1.

```
addi $t0, $0, 7
sw   $t0, 0xFFF4($0) # FFF4 is sign-extended to 0xFFFFFFF4
```

The address decoder asserts *WE1* because the address is 0xFFFFFFF4 and *MemWrite* is TRUE. The value on the *WriteData* bus, 7, is written into the register connected to the input pins of I/O Device 1.

To read from I/O Device 1, the processor performs the following MIPS assembly code.

```
lw $t1, 0xFFF4($0)
```

The address decoder sets $RDsel_{1:0}$ to 01, because it detects the address 0xFFFFFFF4 and *MemWrite* is FALSE. The output of I/O Device 1 passes through the multiplexer onto the *ReadData* bus and is loaded into `$t1` in the processor.

Software that communicates with an I/O device is called a *device driver*. You have probably downloaded or installed device drivers for your printer or other I/O device. Writing a device driver requires detailed knowledge about the I/O device hardware. Other programs call functions in the device driver to access the device without having to understand the low-level device hardware.

The addresses associated with I/O devices are often called I/O *registers* because they may correspond with physical registers in the I/O device like those shown in Figure 8.28.

The next sections of this chapter provide concrete examples of I/O devices. Section 8.6 examines I/O in the context of embedded systems, showing how to use a MIPS-based microcontroller to control many physical devices. Section 8.7 surveys the major I/O systems used in PCs.

8.6 EMBEDDED I/O SYSTEMS

Embedded systems use a processor to control interactions with the physical environment. They are typically built around *microcontroller units* (MCUs) which combine a microprocessor with a set of easy-to-use peripherals such as general-purpose digital and analog I/O pins, serial ports, timers, etc. Microcontrollers are generally inexpensive and are designed to minimize system cost and size by integrating most of the necessary components onto a single chip. Most are smaller and lighter than a dime, consume milliwatts of power, and range in cost from a few dimes up to several dollars. Microcontrollers are classified by the size of data that they operate upon. 8-bit microcontrollers are the smallest and least expensive, while 32-bit microcontrollers provide more memory and higher performance.

For the sake of concreteness, this section will illustrate embedded system I/O in the context of a commercial microcontroller. Specifically, we will focus on the PIC32MX675F512H, a member of Microchip's PIC32-series of microcontrollers based on the 32-bit MIPS microprocessor. The PIC32 family also has a generous assortment of on-chip peripherals and memory so that the complete system can be built with few external components. We selected this family because it has an inexpensive, easy-to-use development environment, it is based on the MIPS architecture studied in this book, and because Microchip is a leading microcontroller vendor that sells more than a billion chips a year. Microcontroller I/O systems are quite similar from one manufacturer to another, so the principles illustrated on the PIC32 can readily be adapted to other microcontrollers.

The rest of this section will illustrate how microcontrollers perform general-purpose digital, analog, and serial I/O. Timers are also commonly used to generate or measure precise time intervals. The section concludes with other interesting peripherals such as displays, motors, and wireless links.

Approximately $16B of microcontrollers were sold in 2011, and the market continues to grow by about 10% a year. Microcontrollers have become ubiquitous and nearly invisible, with an estimated 150 in each home and 50 in each automobile in 2010. The 8051 is a classic 8-bit microcontroller originally developed by Intel in 1980 and now sold by a host of manufacturers. Microchip's PIC16 and PIC18-series are 8-bit market leaders. The Atmel AVR series of microcontrollers has been popularized among hobbyists as the brain of the Arduino platform. Among 32-bit microcontrollers, Renesas leads the overall market, while ARM is a major player in mobile systems including the iPhone. Freescale, Samsung, Texas Instruments, and Infineon are other major microcontroller manufacturers.

8.6.1 PIC32MX675F512H Microcontroller

Figure 8.29 shows a block diagram of a PIC32-series microcontroller. At the heart of the system is the 32-bit MIPS processor. The processor connects via 32-bit busses to Flash memory containing the program and SRAM containing data. The PIC32MX675F512H has 512 KB of Flash and 64 KB of RAM; other flavors with 16 to 512 KB of FLASH and 4 to 128 KB of RAM are available at various price points. High performance peripherals such as USB and Ethernet also communicate directly with the RAM through a bus matrix. Lower performance peripherals including serial ports, timers, and A/D converters share a separate peripheral bus. The chip also contains timing generation circuitry to produce the clocks and voltage sensing circuitry to detect when the chip is powering up or about to lose power.

Figure 8.30 shows the virtual memory map of the microcontroller. All addresses used by the programmer are virtual. The MIPS architecture offers a 32-bit address space to access up to 2^{32} bytes = 4 GB of memory, but only a small fraction of that memory is actually implemented on the chip. The relevant sections from address 0xA0000000-0xBFC02FFF include the RAM, the Flash, and the special function registers (SFRs) used to communicate with peripherals. Note that there is an additional 12 KB of Boot Flash that typically performs some initialization and then jumps to the main program in Flash memory. On reset, the program counter is initialized to the start of the Boot Flash at 0xBFC00000.

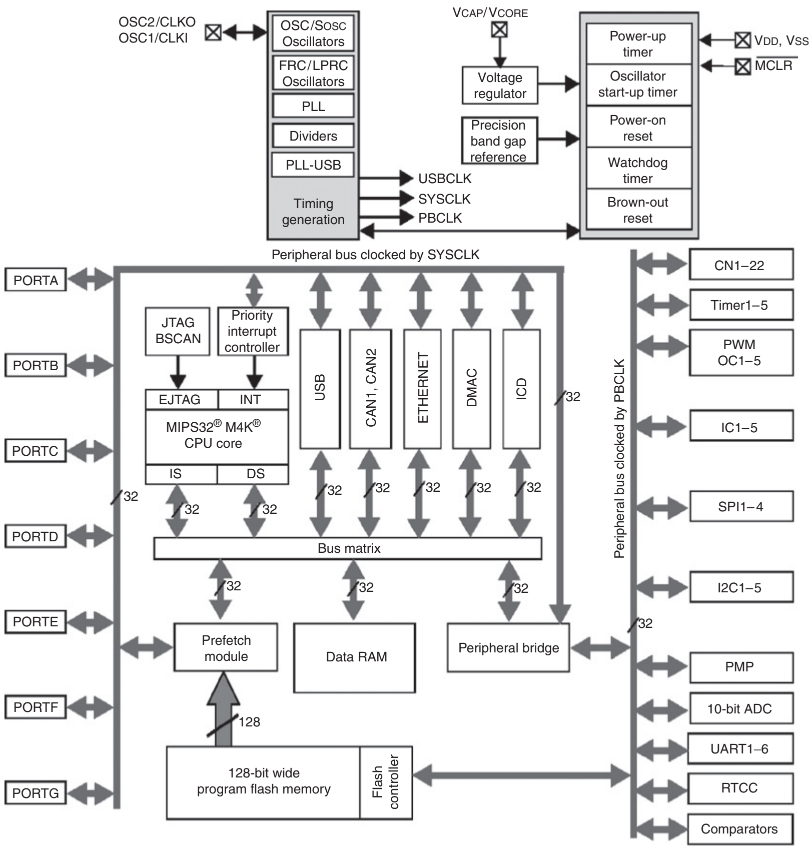

OSC2/CLKO
OSC1/CLKI
OSC/SOSC Oscillators
FRC/LPRC Oscillators
PLL
Dividers
PLL-USB
Timing generation
USBCLK
SYSCLK
PBCLK
VCAP/VCORE
Voltage regulator
Precision band gap reference
Power-up timer
Oscillator start-up timer
Power-on reset
Watchdog timer
Brown-out reset
VDD, VSS
MCLR
Peripheral bus clocked by SYSCLK
PORTA
PORTB
PORTC
PORTD
PORTE
PORTF
PORTG
JTAG BSCAN
Priority interrupt controller
EJTAG
INT
MIPS32® M4K® CPU core
IS
DS
USB
CAN1, CAN2
ETHERNET
DMAC
ICD
32
Bus matrix
Prefetch module
Data RAM
Peripheral bridge
128
128-bit wide program flash memory
Flash controller
Peripheral bus clocked by PBCLK
CN1–22
Timer1–5
PWM OC1–5
IC1–5
SPI1–4
I2C1–5
PMP
10-bit ADC
UART1–6
RTCC
Comparators

Figure 8.29 **PIC32MX675F512H block diagram** (© 2012 Microchip Technology Inc.; reprinted with permission.)

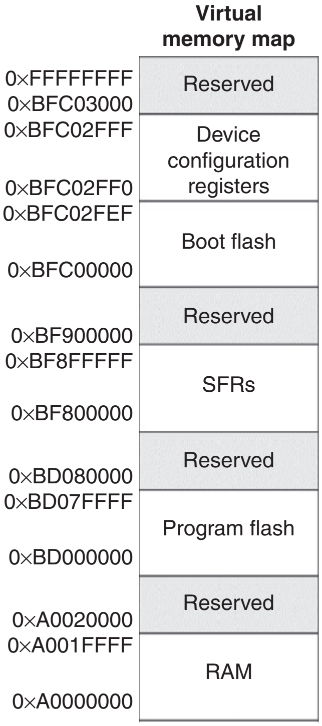

Virtual memory map
0×FFFFFFFF
Reserved
0×BFC03000
0×BFC02FFF
Device configuration registers
0×BFC02FF0
0×BFC02FEF
Boot flash
0×BFC00000
Reserved
0×BF900000
0×BF8FFFFF
SFRs
0×BF800000
Reserved
0×BD080000
0×BD07FFFF
Program flash
0×BD000000
Reserved
0×A0020000
0×A001FFFF
RAM
0×A0000000

Figure 8.30 **PIC32 memory map** (© 2012 Microchip Technology Inc.; reprinted with permission.)

Figure 8.31 shows the pinout of the microcontroller. Pins include power and ground, clock, reset, and many I/O pins that can be used for general-purpose I/O and/or special-purpose peripherals. Figure 8.32 shows a photograph of the microcontroller in a 64-pin Thin Quad Flat Pack (TQFP) with pins along the four sides spaced at 20 mil (0.02 inch) intervals. The microcontroller is also available in a 100-pin package with more I/O pins; that version has a part number ending with an L instead of an H.

Figure 8.33 shows the microcontroller connected in a minimal operational configuration with a power supply, an external clock, a reset switch, and a jack for a programming cable. The PIC32 and external circuitry are mounted on a printed circuit board; this board may be an actual product (e.g. a toaster controller), or a development board facilitating easy access to the chip during testing.

The LTC1117-3.3 regulator accepts an input of 4.5-12 V (e.g., from a wall transformer, battery, or external power supply) and drops it down to a steady 3.3 V required at the power supply pins. The PIC32 has multiple

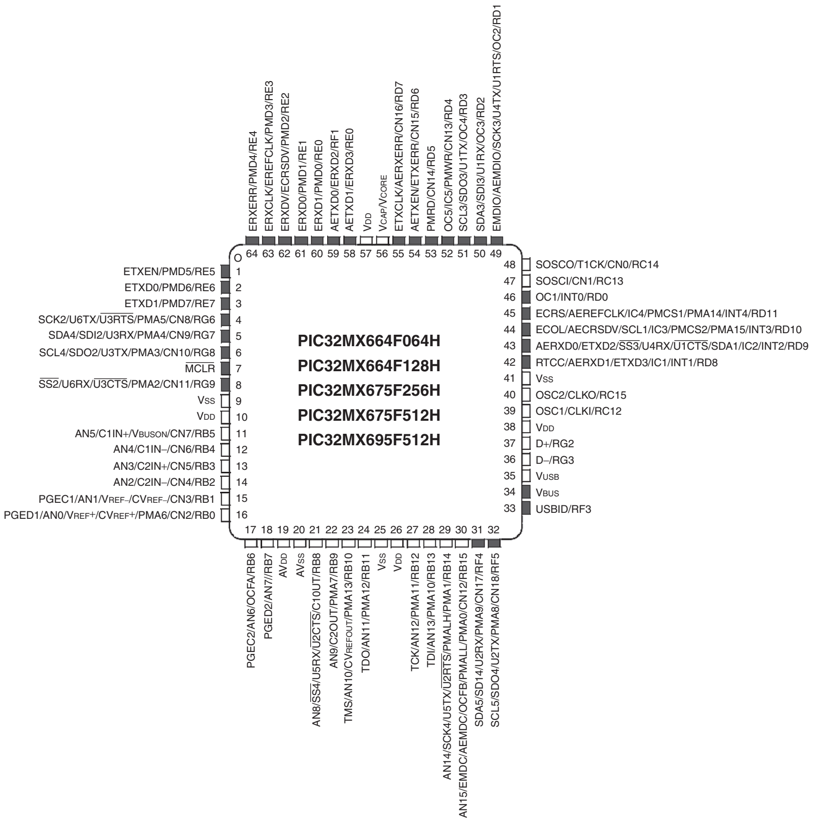

PIC32MX664F064H
PIC32MX664F128H
PIC32MX675F256H
PIC32MX675F512H
PIC32MX695F512H
1 ETXEN/PMD5/RE5
2 ETXD0/PMD6/RE6
3 ETXD1/PMD7/RE7
4 SCK2/U6TX/U3RTS/PMA5/CN8/RG6
5 SDA4/SDI2/U3RX/PMA4/CN9/RG7
6 SCL4/SDO2/U3TX/PMA3/CN10/RG8
7 MCLR
8 SS2/U6RX/U3CTS/PMA2/CN11/RG9
9 VSS
10 VDD
11 AN5/C1IN+/VBUSON/CN7/RB5
12 AN4/C1IN–/CN6/RB4
13 AN3/C2IN+/CN5/RB3
14 AN2/C2IN–/CN4/RB2
15 PGEC1/AN1/VREF–/CVREF–/CN3/RB1
16 PGED1/AN0/VREF+/CVREF+/PMA6/CN2/RB0
17 PGEC2/AN6/OCFA/RB6
18 PGED2/AN7//RB7
19 AVDD
20 AVSS
21 AN8/SS4/U5RX/U2CTS/C1OUT/RB8
22 AN9/C2OUT/PMA7/RB9
23 TMS/AN10/CVREFOUT/PMA13/RB10
24 TDO/AN11/PMA12/RB11
25 VSS
26 VDD
27 TCK/AN12/PMA11/RB12
28 TDI/AN13/PMA10/RB13
29 AN14/SCK4/U5TX/U2RTS/PMALH/PMA1/RB14
30 AN15/EMDC/AEMDC/OCFB/PMALL/PMA0/CN12/RB15
31 SDA5/SDI4/U2RX/PMA9/CN17/RF4
32 SCL5/SDO4/U2TX/PMA8/CN18/RF5
33 USBID/RF3
34 VBUS
35 VUSB
36 D–/RG3
37 D+/RG2
38 VDD
39 OSC1/CLKI/RC12
40 OSC2/CLKO/RC15
41 VSS
42 RTCC/AERXD1/ETXD3/IC1/INT1/RD8
43 AERXD0/ETXD2/SS3/U4RX/U1CTS/SDA1/IC2/INT2/RD9
44 ECOL/AECRSDV/SCL1/IC3/PMCS2/PMA15/INT3/RD10
45 ECRS/AEREFCLK/IC4/PMCS1/PMA14/INT4/RD11
46 OC1/INT0/RD0
47 SOSCI/CN1/RC13
48 SOSCO/T1CK/CN0/RC14
49 EMDIO/AEMDIO/SCK3/U4TX/U1RTS/OC2/RD1
50 SDA3/SDI3/U1RX/OC3/RD2
51 SCL3/SDO3/U1TX/OC4/RD3
52 OC5/IC5/PMWR/CN13/RD4
53 PMRD/CN14/RD5
54 AETXEN/ETXERR/CN15/RD6
55 ETXCLK/AERXERR/CN16/RD7
56 VCAP/VCORE
57 VDD
58 AETXD1/ERXD3/RE0
59 AETXD0/ERXD2/RF1
60 ERXD1/PMD0/RE0
61 ERXD0/PMD1/RE1
62 ERXDV/ECRSDV/PMD2/RE2
63 ERXCLK/EREFCLK/PMD3/RE3
64 ERXERR/PMD4/RE4

Figure 8.31 **PIC32MX6xxFxxH pinout. Black pins are 5 V-tolerant.**
(© 2012 Microchip Technology Inc.; reprinted with permission.)

VDD and GND pins to reduce power supply noise by providing a low-impedance path. An assortment of bypass capacitors provide a reservoir of charge to keep the power supply stable in the event of sudden changes in current demand. A bypass capacitor also connects to the VCORE pin, which serves an internal 1.8 V voltage regulator. The PIC32 typically draws 1–2 mA/MHz of current from the power supply. For example, at 80 MHz, the maximum power dissipation is $P = VI = (3.3 \text{ V})(120 \text{ mA}) = 0.396 \text{ W}$. The 64-pin TQFP has a thermal resistance of 47°C/W, so the chip may heat up by about 19°C if operated without a heat sink or cooling fan.

An external oscillator running up to 50 MHz can be connected to the clock pin. In this example circuit, the oscillator is running at 40.000 MHz. Alternatively, the microcontroller can be programmed to use an internal oscillator running at 8.00 MHz ± 2%. This is much less precise in frequency, but may be good enough. The peripheral bus clock, PBCLK, for the I/O devices (such as serial ports, A/D converter, timers) typically

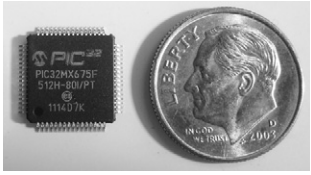

PIC32
PIC32MX675F
512H-80I/PT
1114D7K
LIBERTY
IN GOD WE TRUST
2003

Figure 8.32 **PIC32 in 64-pin TQFP package**

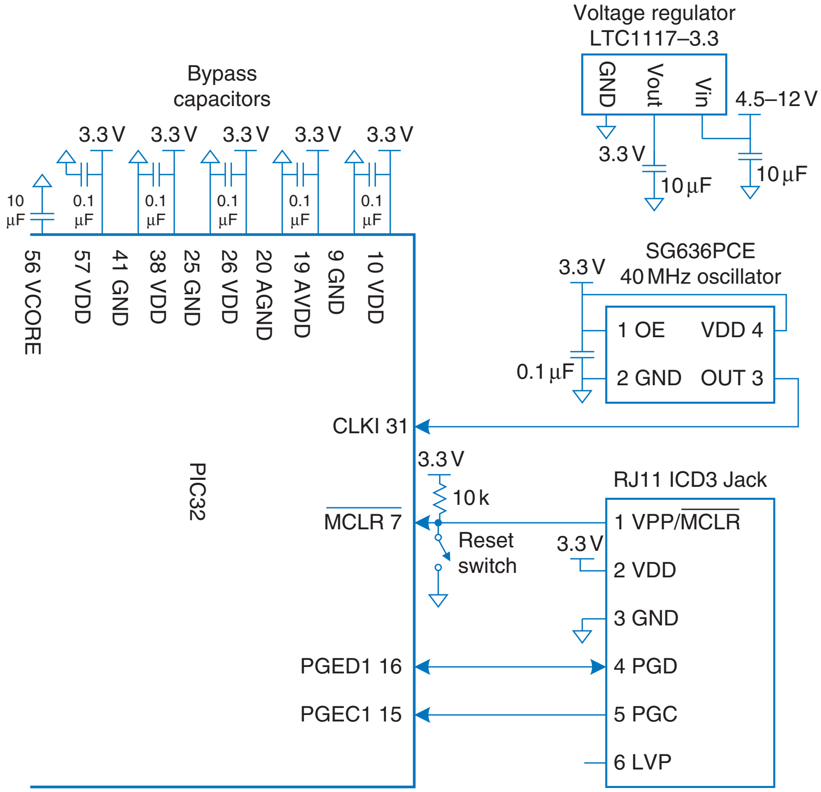

Voltage regulator
LTC1117–3.3
GND
Vout
Vin
4.5–12 V
3.3 V
10 µF
10 µF
Bypass capacitors
3.3 V
3.3 V
3.3 V
3.3 V
3.3 V
10 µF
0.1 µF
0.1 µF
0.1 µF
0.1 µF
0.1 µF
56 VCORE
57 VDD
41 GND
38 VDD
25 GND
26 VDD
20 AGND
19 AVDD
9 GND
10 VDD
3.3 V
SG636PCE
40 MHz oscillator
1 OE
VDD 4
2 GND
OUT 3
0.1 µF
CLKI 31
PIC32
3.3 V
10 k
MCLR 7
Reset switch
RJ11 ICD3 Jack
1 VPP/MCLR
3.3 V
2 VDD
3 GND
PGED1 16
4 PGD
PGEC1 15
5 PGC
6 LVP

Figure 8.33 **PIC32 basic operational schematic**

runs at a fraction (e.g., half) of the speed of the main system clock SYSCLK. This clocking scheme can be set by configuration bits in the MPLAB development software or by placing the following lines of code at the start of your C program.

```
#pragma config FPBDIV = DIV_2  // peripherals operate at half
                               // sysckfreq (20 MHz)
#pragma config POSCMOD = EC    // configure primary oscillator in
                               // external clock mode
#pragma config FNOSC = PRI     // select the primary oscillator
```

It is always handy to provide a reset button to put the chip into a known initial state of operation. The reset circuitry consists of a push-button switch and a resistor connected to the reset pin, $\overline{MCLR}$. The reset pin is active-low, indicating that the processor resets if the pin is at 0. When the button is not pressed, the switch is open and the resistor pulls the reset pin to 1, allowing normal operation. When the button is pressed, the switch closes and pulls the reset pin down to 0, forcing the processor to reset. The PIC32 also resets itself automatically when power is turned on.

The peripheral bus clock PBCLK can be configured to operate at the same speed as the main system clock, or at half, a quarter, or an eighth of the speed. Early PIC32 microcontrollers were limited to a maximum of half-speed operation because some peripherals were too slow, but most current products can run at full speed. If you change the PBCLK speed, you'll need to adjust parts of the sample code in this section such as the number of PBCLK ticks to measure a certain amount of time.

The easiest way to program the microcontroller is with a Microchip *In Circuit Debugger* (ICD) 3, affectionately known as a puck. The ICD3, shown in Figure 8.34, allows the programmer to communicate with the PIC32 from a PC to download code and to debug the program. The ICD3 connects to a USB port on the PC and to a six-pin RJ-11 modular connector on the PIC32 development board. The RJ-11 connector is the familiar connector used in United States telephone jacks. The ICD3 communicates with the PIC32 over a 2-wire In-Circuit Serial Programming interface with a clock and a bidirectional data pin. You can use Microchip's free MPLAB Integrated Development Environment (IDE) to write your programs in assembly language or C, debug them in simulation, and download and test them on a development board by means of the ICD.

POWER
ACTIVE
STATUS
In-Circuit Debugger
MPLAB
ICD 3
MICROCHIP

Figure 8.34 **Microchip ICD3**

Most of the pins of the PIC32 microcontroller default to function as general-purpose digital I/O pins. Because the chip has a limited number of pins, these same pins are shared for special-purpose I/O functions such as serial ports, analog-to-digital converter inputs, etc., that become active when the corresponding peripheral is enabled. It is the responsibility of the programmer to use each pin for only one purpose at any given time. The remainder of Section 8.6 explores the microcontroller I/O functions in detail.

Microcontroller capabilities go far beyond what can be covered in the limited space of this chapter. See the manufacturer's datasheet for more detail. In particular, the *PIC32MX5XX/6XX/7XX Family Data Sheet* and *PIC32 Family Reference Manual* from Microchip are authoritative and tolerably readable.

8.6.2 General-Purpose Digital I/O

General-purpose I/O (GPIO) pins are used to read or write digital signals. For example, Figure 8.35 shows eight light-emitting diodes (LEDs) and four switches connected to a 12-bit GPIO port. The schematic indicates the name and pin number of each of the port's 12 pins; this tells the programmer the function of each pin and the hardware designer what connections to physically make. The LEDs are wired to glow when driven with a 1 and turn off when driven with a 0. The switches are wired to produce a 1 when closed and a 0 when open. The microcontroller can use the port both to drive the LEDs and to read the state of the switches.

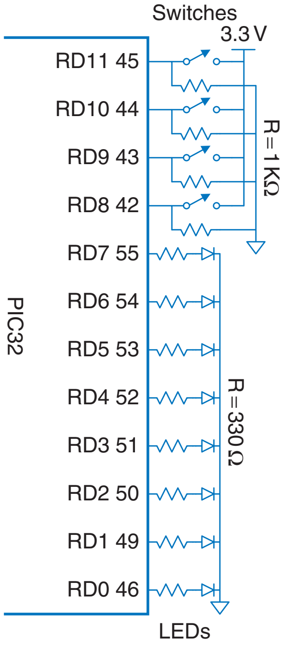


Figure 8.35 **LEDs and switches connected to 12-bit GPIO port D**

The PIC32 organizes groups of GPIOs into ports that are read and written together. Our PIC32 calls these ports RA, RB, RC, RD, RE, RF, and RG; they are referred to as simply port A, port B, etc. Each port may have up to 16 GPIO pins, although the PIC32 doesn't have enough pins to provide that many signals for all of its ports.

Each port is controlled by two registers: TRISx and PORTx, where x is a letter (A-G) indicating the port of interest. The TRISx registers determine whether the pin of the port is an input or an output, while the PORTx registers indicate the value read from an input or driven to an output. The 16 least significant bits of each register correspond to the sixteen pins of the GPIO port. When a given bit of the TRISx register is 0, the pin is an output, and when it is 1, the pin is an input. It is prudent to leave unused GPIO pins as inputs (their default state) so that they are not inadvertently driven to troublesome values.

Each register is memory-mapped to a word in the Special Function Registers portion of virtual memory (0xBF80000-BF8FFFF). For example, TRISD is at address 0xBF8860C0 and PORTD is at address 0xBF8860D0. The p32xxxx.h header file declares these registers as 32-bit unsigned integers. Hence, the programmer can access them by name rather than having to look up the address.

Example 8.18 GPIO FOR SWITCHES AND LEDs

Write a C program to read the four switches and turn on the corresponding bottom four LEDs using the hardware in Figure 8.35.

Solution: Configure TRISD so that pins RD[7:0] are outputs and RD[11:8] are inputs. Then read the switches by examining pins RD[11:8], and write this value back to RD[3:0] to turn on the appropriate LEDs.

```
#include <p32xxxx.h>

int main(void) {
  int switches;

  TRISD = 0xFF00;                    // set RD[7:0] to output,
                                     // RD[11:8] to input
  while (1) {
    switches = (PORTD >> 8) & 0xF;   // Read and mask switches from
                                     // RD[11:8]
    PORTD = switches;                // display on the LEDs
  }
}
```

Example 8.18 writes the entire register at once. It is also possible to access individual bits. For example, the following code copies the value of the first switch to the first LED.

```
PORTDbits.RD0 = PORTDbits.RD8;
```

In the context of bit manipulation, “setting” means writing to 1 and “clearing” means writing to 0.

Each port also has corresponding SET and CLR registers that can be written with a mask indicating which bits to set or clear. For example,

```
PORTDSET = 0b0101;
PORTDCLR = 0b1000;
```

Table 8.5 PIC32MX5xx/6xx/7xx GPIO pins

Port	64-pin QFN/TQFP	100-pin TQFP, 121-pin XBGA
RA	None	15:14, 10:9, 7:0
RB	15:0	15:0
RC	15:12	15:12, 4:0
RD	11:0	15:0
RE	7:0	9:0
RF	5:3, 1:0	13:12, 8, 5:0
RG	9:6, 3:2	15:12, 9:6, 3:0

sets the first and third bits of PORTD and clears the fourth bit. If the bottom four bits of PORTD had been 1110, they would become 0111.

The number of GPIO pins available depends on the size of the package. Table 8.5 summarizes which pins are available in various packages. For example, a 100-pin TQFP provides pins RA[15:14], RA[10:9], and RA[7:0] of port A. Beware that RG[3:2] are input only. Also, RB[15:0] are shared as analog input pins, and the other pins have multiple functions as well.

The logic levels are LVCMOS-compatible. Input pins expect logic levels of $V_{IL} = 0.15V_{DD}$ and $V_{IH} = 0.8V_{DD}$, or 0.5 V and 2.6 V assuming V_{DD} of 3.3V. Output pins produce V_{OL} of 0.4 V and V_{OH} of 2.4 V as long as the output current I_{out} does not exceed a paltry 7 mA.

8.6.3 Serial I/O

If a microcontroller needs to send more bits than the number of free GPIO pins, it must break the message into multiple smaller transmissions. In each step, it can send either one bit or several bits. The former is called *serial* I/O and the latter is called *parallel* I/O. Serial I/O is popular because it uses few wires and is fast enough for many applications. Indeed, it is so popular that multiple standards for serial I/O have been established and the PIC32 has dedicated hardware to easily send data via these standards. This section describes the Serial Peripheral Interface (SPI) and Universal Asynchronous Receiver/Transmitter (UART) standard protocols.

Other common serial standards include Inter-Integrated Circuit (I^2C), Universal Serial Bus (USB), and Ethernet. I^2C (pronounced "I squared C") is a 2-wire interface with a clock and a bidirectional data pin; it is used in a fashion similar to SPI. USB and Ethernet are more complex, high-performance standards described in Sections 8.7.1 and 8.7.4, respectively.

Serial Peripheral Interface (SPI)

SPI (pronounced "S-P-I") is a simple synchronous serial protocol that is easy to use and relatively fast. The physical interface consists of three pins: Serial Clock (SCK), Serial Data Out (SDO), and Serial Data In (SDI). SPI connects a *master* device to a *slave* device, as shown in Figure 8.36(a). The master produces the clock. It initiates communication by sending a series of clock pulses on SCK. If it wants to send data to the slave, it puts the data on SDO, starting with the most significant bit. The slave may simultaneously respond by putting data on the master's SDI. Figure 8.36(b) shows the SPI waveforms for an 8-bit data transmission.

The PIC32 has up to four SPI ports unsurprisingly named SPI1-SPI4. Each can operate as a master or slave. This section describes master mode operation, but slave mode is similar. To use an SPI port, the PIC® program must first configure the port. It can then write data to a register; the data is transmitted serially to the slave. Data received from the slave is collected in another register. When the transmission is complete, the PIC32 can read the received data.

Each SPI port is associated with four 32-bit registers: SPIxCON, SPIxSTAT, SPIxBRG, and SPIxBUF. For example, SPI1CON is the control register for SPI port 1. It is used to turn the SPI ON and set attributes such as the number of bits to transfer and the polarity of the clock. Table 8.6 lists the names and functions of all the bits of the CON registers. All have a default value of 0 on reset. Most of the functions, such as framing, enhanced buffering, slave select signals, and

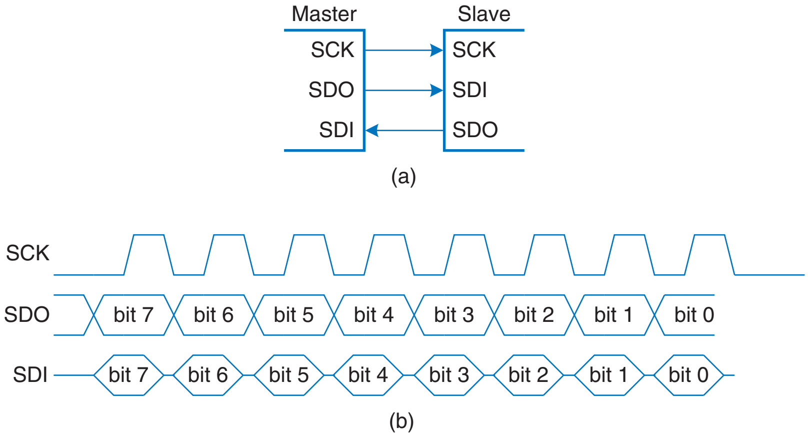


Figure 8.36 **SPI connection and master waveforms**

Table 8.6 **SPIxCON register fields**

Bits	Name	Function
31	`FRMEN`	1: Enable framing
30	`FRMSYNC`	Frame sync pulse direction control
29	`FRMPOL`	Frame sync polarity (1 = active high)
28	`MSSEN`	1: Enable slave select generation in master mode
27	`FRMSYPW`	Frame sync pulse width bit (1 = 1 word wide, 0 = 1 clock wide)
26:24	`FRMCNT[2:0]`	Frame sync pulse counter (frequency of sync pulses)
23	`MCLKSEL`	Master clock select (1 = master clock, 0 = peripheral clock)
22:18	`unused`	
17	`SPIFE`	Frame sync pulse edge select
16	`ENHBUF`	1: Enable enhanced buffering
15	`ON`	1: SPI ON
14	`unused`	
13	`SIDL`	1: Stop SPI when CPU is in idle mode
12	`DISSDO`	1: disable SDO pin
11	`MODE32`	1: 32-bit transfers
10	`MODE16`	1: 16-bit transfers
9	`SMP`	Sample phase (see Figure 8.39)
8	`CKE`	Clock edge (see Figure 8.39)
7	`SSEN`	1: Enable slave select
6	`CKP`	Clock polarity (see Figure 8.39)
5	`MSTEN`	1: Enable master mode
4	`DISSDI`	1: disable SDI pin
3:2	`STXISEL[1:0]`	Transmit buffer interrupt mode
1:0	`SRXISEL[1:0]`	Receive buffer interrupt mode

interrupts are not used in this section but can be found in the datasheet. STAT is the status register indicating, for example, whether the receive register is full. Again, the details of this register are also fully described in the PIC32 datasheet.

The serial clock can be configured to toggle at half the peripheral clock rate or less. SPI has no theoretical limit on the data rate and can readily operate at tens of MHz on a printed circuit board, though it may experience noise problems running above 1 MHz using wires on a breadboard. BRG is the baud rate register that sets the speed of SCK relative to the peripheral clock according to the formula:

$$f_{SPI} = \frac{f_{peripheral\_clock}}{2 \times (\text{BRG} + 1)} \tag{8.3}$$

BUF is the data buffer. Data written to BUF is transferred over the SPI port on the SDO pin, and the received data from the SDI pin can be found by reading BUF after the transfer is complete.

To prepare the SPI in master mode, first turn it OFF by clearing bit 15 of the CON register (the ON bit) to 0. Clear anything that might be in the receive buffer by reading the BUF register. Set the desired baud rate by writing the BRG register. For example, if the peripheral clock is 20 MHz and the desired baud rate is 1.25 MHz, set BRG to $[20/(2 \times 1.25)] - 1 = 7$. Put the SPI in master mode by setting bit 5 of the CON register (MSTEN) to 1. Set bit 8 of the CON register (CKE) so that SDO is centered on the rising edge of the clock. Finally, turn the SPI back ON by setting the ON bit of the CON register.

To send data to the slave, write the data to the BUF register. The data will be transmitted serially, and the slave will simultaneously send data back to the master. Wait until bit 11 of the STAT register (the SPIBUSY bit) becomes 0 indicating that the SPI has completed its operation. Then the data received from the slave can be read from BUF.

The SPI port on a PIC32 is highly configurable so that it can talk to a wide variety of serial devices. Unfortunately, this leads to the possibility of incorrectly configuring the port and getting garbled data transmissions. The relative timing of the clock and data signals are configured with three CON register bits called CKP, CKE, and SMP. By default, these bits are 0, but the timing in Figure 8.36(b) uses CKE = 1. The master changes SDO on the falling edge of SCK, so the slave should sample the value on the rising edge using a positive edge-triggered flip-flop. The master expects SDI to be stable around the rising edge of SCK, so the slave should change it on the falling edge, as shown in the timing diagram. The MODE32 and MODE16 bits of the CON register specify whether a 32- or 16-bit word should be sent; these both default to 0 indicating an 8-bit transfer.

Baud rate gives the signaling rate, measured in symbols per second, whereas bit rate gives the data rate, measured in bits per second. The signaling we've discussed in this text is 2-level signaling, where each symbol represents a bit. However, multi-level signaling can send multiple bits per symbol; for example, 4-level signaling sends two bits per symbol. In that case, the bit rate is twice the baud rate. In a simple system where each symbol is a bit and each symbol represents data, the baud rate is equal to the bit rate. Some signaling conventions require overhead bits in addition to the data (see Section 8.6.3.2 on UARTs). For example, a two-level signaling system that uses two overhead bits for each 8 bits of data with a baud rate of 9600 has a bit rate of (9600 symbols/second) × (8 bits/10 symbols) = 7680 bits/second.

SPI always sends data in both directions on each transfer. If the system only needs unidirectional communication, it can ignore the unwanted data. For example, if the master only needs to send data to the slave, the byte received from the slave can be ignored. If the master only needs to receive data from the slave, it must still trigger the SPI communication by sending an arbitrary byte that the slave will ignore. It can then read the data received from the slave.

Example 8.19 SENDING AND RECEIVING BYTES OVER SPI

Design a system to communicate between a PIC® master and an FPGA slave over SPI. Sketch a schematic of the interface. Write the C code for the microcontroller to send the character 'A' and receive a character back. Write HDL code for an SPI slave on the FPGA. How could the slave be simplified if it only needs to receive data?

Solution: Figure 8.37 shows the connection between the devices using SPI port 2. The pin numbers are obtained from the component datasheets (e.g., Figure 8.31). Notice that both the pin numbers and signal names are shown on the diagram to indicate both the physical and logical connectivity. These pins are also used by GPIO port RG[8:6]. When the SPI is enabled, these bits of port G cannot be used for GPIO.

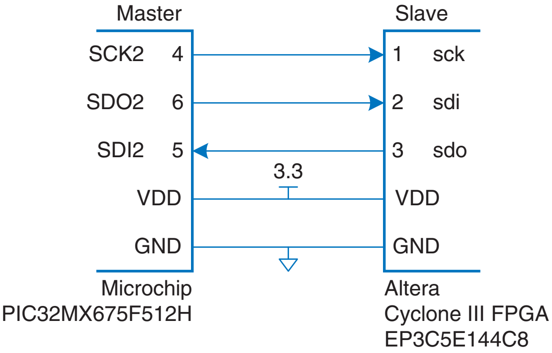


Figure 8.37 **SPI connection between PIC32 and FPGA**

The C code below initializes the SPI and then sends and receives a character.

```
#include <p32xxxx.h>

void initspi(void) {
  char junk;

  SPI2CONbits.ON = 0;    // disable SPI to reset any previous state
  junk = SPI2BUF;        // read SPI buffer to clear the receive buffer
  SPI2BRG = 7;
  SPI2CONbits.MSTEN = 1; // enable master mode
  SPI2CONbits.CKE = 1;   // set clock-to-data timing
  SPI2CONbits.ON = 1;    // turn SPI on
}

char spi_send_receive(char send) {
  SPI2BUF = send;                // send data to slave
  while (SPI2STATbits.SPIBUSY); // wait until SPI transmission complete
  return SPI2BUF;                // return received data
}
```

```
int main(void) {
  char received;

  initspi();                         // initialize the SPI port
  received = spi_send_receive('A');  // send letter A and receive byte
                                     // back from slave
}
```

The HDL code for the FPGA is listed below and a block diagram with timing is shown in Figure 8.38. The FPGA uses a shift register to hold the bits that have been received from the master and the bits that remain to be sent to the master. On the first rising `sck` edge after reset and each 8 cycles thereafter, a new byte from `d` is loaded into the shift register. On each subsequent cycle, a bit is shifted in on the FPGA's `sdi` and a bit is shifted out of the FPGA's `sdo`. `sdo` is delayed until the falling edge of `sck` so that it can be sampled by the master on the next rising edge. After 8 cycles, the byte received can be found in `q`.

```
module spi_slave(input  logic       sck,   // from master
                 input  logic       sdi,   // from master
                 output logic       sdo,   // to master
                 input  logic       reset, // system reset
                 input  logic [7:0] d,     // data to send
                 output logic [7:0] q);    // data received
```

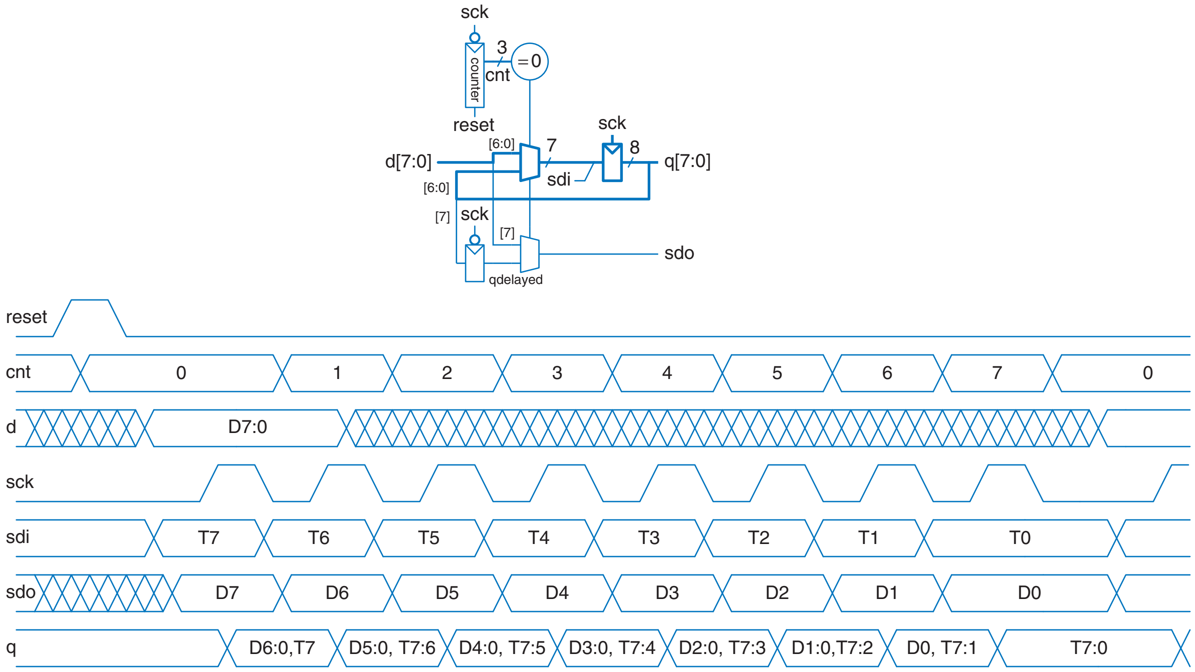

sck
3
=0
counter
cnt
reset
sck
[6:0]
7
8
d[7:0]
q[7:0]
sdi
[6:0]
[7]
sck
[7]
sdo
qdelayed
reset
cnt
0
1
2
3
4
5
6
7
0
d
D7:0
sck
sdi
T7
T6
T5
T4
T3
T2
T1
T0
sdo
D7
D6
D5
D4
D3
D2
D1
D0
q
D6:0,T7
D5:0, T7:6
D4:0, T7:5
D3:0, T7:4
D2:0, T7:3
D1:0,T7:2
D0, T7:1
T7:0

Figure 8.38 SPI slave circuitry and timing

```
  logic [2:0] cnt;
  logic       qdelayed;

  // 3-bit counter tracks when full byte is transmitted
  always_ff @(negedge sck, posedge reset)
    if (reset) cnt = 0;
    else       cnt = cnt + 3'b1;

  // loadable shift register
  // loads d at the start, shifts sdi into bottom on each step
  always_ff @(posedge sck)
    q <= (cnt == 0) ? {d[6:0], sdi} : {q[6:0], sdi};

  // align sdo to falling edge of sck
  // load d at the start
  always_ff @(negedge sck)
    qdelayed = q[7];
  assign sdo = (cnt == 0) ? d[7] : qdelayed;
endmodule
```

If the slave only needs to receive data from the master, it reduces to a simple shift register given in the following HDL code.

```
module spi_slave_receive_only(input  logic       sck, // from master
                              input  logic       sdi, // from master
                              output logic [7:0] q);  // data received

  always_ff @(posedge sck)
    q <= {q[6:0], sdi}; // shift register
endmodule
```

Sometimes it is necessary to change the configuration bits to communicate with a device that expects different timing. When CKP = 1, SCK is inverted. When CKE = 0, the clocks toggle half a cycle earlier relative to the data. When SAMPLE = 1, the master samples SDI half a cycle later (and the slave should ensure it is stable at that time). These modes are shown in Figure 8.39. Be aware that different SPI products may use different names and polarities for these options; check the waveforms carefully for your device. It can also be helpful to examine SCK, SDO, and SDI on an oscilloscope if you are having communication difficulties.

Universal Asynchronous Receiver Transmitter (UART)

A UART (pronounced "you-art") is a serial I/O peripheral that communicates between two systems without sending a clock. Instead, the systems must agree in advance about what data rate to use and must each locally generate its own clock. Although these system clocks may have a small frequency error and an unknown phase relationship, the UART manages reliable asynchronous communication. UARTs are used in protocols such

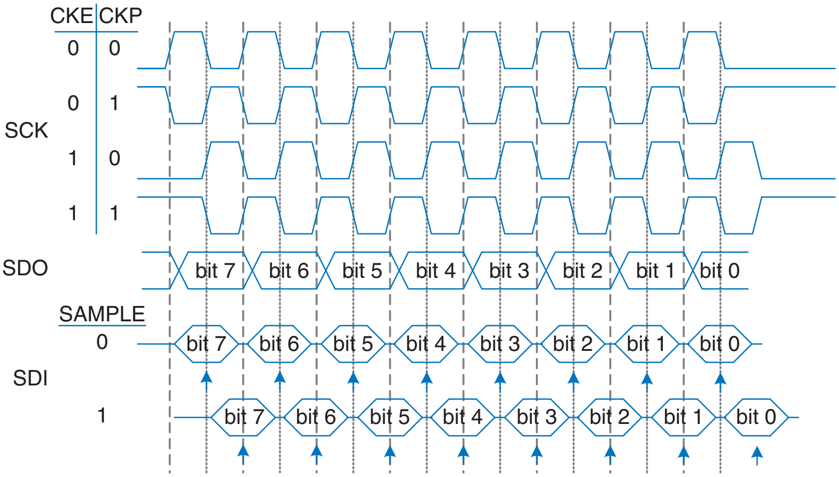

CKE
CKP
0 0
0 1
SCK
1 0
1 1
SDO
bit 7
bit 6
bit 5
bit 4
bit 3
bit 2
bit 1
bit 0
SAMPLE
0
SDI
1

Figure 8.39 **Clock and data timing controlled by CKE, CKP, and SAMPLE**

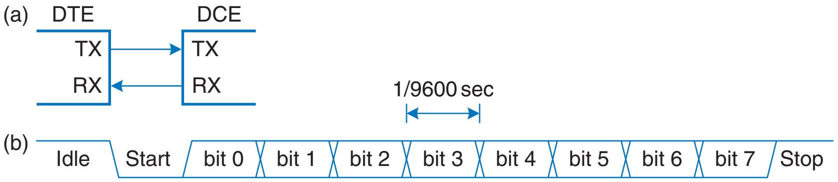

(a)
DTE
DCE
TX
RX
TX
RX
1/9600 sec
(b)
Idle
Start
bit 0
bit 1
bit 2
bit 3
bit 4
bit 5
bit 6
bit 7
Stop

Figure 8.40 **Asynchronous serial link**

as RS-232 and RS-485. For example, computer serial ports use the RS-232C standard, introduced in 1969 by the Electronic Industries Association. The standard originally envisioned connecting *Data Terminal Equipment* (DTE) such as a mainframe computer to *Data Communication Equipment* (DCE) such as a modem. Although a UART is relatively slow compared to SPI, the standards have been around for so long that they remain important today.

Figure 8.40(a) shows an asynchronous serial link. The DTE sends data to the DCE over the TX line and receives data back over the RX line. Figure 8.40(b) shows one of these lines sending a character at a data rate of 9600 baud. The line idles at a logic '1' when not in use. Each character is sent as a start bit (0), 7-8 data bits, an optional parity bit, and one or more stop bits (1's). The UART detects the falling transition from idle to start to lock on to the transmission at the appropriate time. Although seven data bits is sufficient to send an ASCII character, eight bits are normally used because they can convey an arbitrary byte of data.

An additional bit, the *parity* bit, can also be sent, allowing the system to detect if a bit was corrupted during transmission. It can be configured as *even* or *odd*; even parity means that the parity bit is chosen such that the total collection of data and parity has an even number of 1's; in other words, the parity bit is the XOR of the data bits. The receiver can then check if an even number of 1's was received and signal an error if not. Odd parity is the reverse and is hence the XNOR of the data bits.

A common choice is 8 data bits, no parity, and 1 stop bit, making a total of 10 symbols to convey an 8-bit character of information. Hence, signaling rates are referred to in units of baud rather than bits/sec. For example, 9600 baud indicates 9600 symbols/sec, or 960 characters/sec, giving a data rate of $960 \times 8 = 7680$ data bits/sec. Both systems must be configured for the appropriate baud rate and number of data, parity, and stop bits or the data will be garbled. This is a hassle, especially for nontechnical users, which is one of the reasons that the Universal Serial Bus (USB) has replace UARTs in personal computer systems.

Typical baud rates include 300, 1200, 2400, 9600, 14400, 19200, 38400, 57600, and 115200. The lower rates were used in the 1970's and 1980's for modems that sent data over the phone lines as a series of tones. In contemporary systems, 9600 and 115200 are two of the most common baud rates; 9600 is encountered where speed doesn't matter, and 115200 is the fastest standard rate, though still slow compared to other modern serial I/O standards.

The RS-232 standard defines several additional signals. The Request to Send (RTS) and Clear to Send (CTS) signals can be used for *hardware handshaking*. They can be operated in either of two modes. In *flow control* mode, the DTE clears RTS to 0 when it is ready to accept data from the DCE. Likewise, the DCE clears CTS to 0 when it is ready to receive data from the DTE. Some datasheets use an overbar to indicate that they are active-low. In the older *simplex* mode, the DTE clears RTS to 0 when it is ready to transmit. The DCE replies by clearing CTS when it is ready to receive the transmission.

Some systems, especially those connected over a telephone line, also use Data Terminal Ready (DTR), Data Carrier Detect (DCD), Data Set Ready (DSR), and Ring Indicator (RI) to indicate when equipment is connected to the line.

The original standard recommended a massive 25-pin DB-25 connector, but PCs streamlined to a male 9-pin DE-9 connector with the pinout shown in Figure 8.42(a). The cable wires normally connect straight across as shown in Figure 8.42(b). However, when directly connecting two DTEs, a *null modem* cable shown in Figure 8.42(c) may be needed to swap RX and TX and complete the handshaking. As a final insult, some connectors are male and some are female. In summary, it can take a large box of cables and a certain amount of guess-work to connect two systems over RS-232, again explaining the shift to USB. Fortunately, embedded systems typically use a simplified 3- or 5-wire setup consisting of GND, TX, RX, and possibly RTS and CTS.

In the 1950s through 1970s, early hackers calling themselves *phone phreaks* learned to control the phone company switches by whistling appropriate tones. A 2600 Hz tone produced by a toy whistle from a Cap'n Crunch cereal box (Figure 8.41) could be exploited to place free long-distance and international calls.

CAP'N
CRUNCH
BO'SUN WHISTLE

Figure 8.41 Cap'n Crunch Bosun Whistle. Photograph by Evrim Sen, reprinted with permission.

Handshaking refers to the negotiation between two systems; typically, one system signals that it is ready to send or receive data, and the other system acknowledges that request.

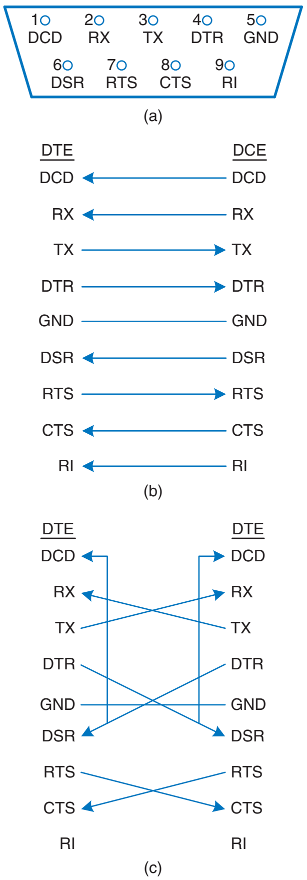


Figure 8.42 DE-9 male cable (a) pinout, (b) standard wiring, and (c) null modem wiring

RS-232 represents a 0 electrically with 3 to 15 V and a 1 with −3 to −15 V; this is called *bipolar* signaling. A transceiver converts the digital logic levels of the UART to the positive and negative levels expected by RS-232, and also provides electrostatic discharge protection to protect the serial port from damage when the user plugs in a cable. The MAX3232E is a transceiver compatible with both 3.3 and 5 V digital logic. It contains a charge pump that, in conjunction with external capacitors, generates ± 5 V outputs from a single low-voltage power supply.

The PIC32 has six UARTs named U1-U6. As with SPI, the PIC® program must first configure the port. Unlike SPI, reading and writing can occur independently because either system may transmit without receiving and vice versa. Each UART is associated with five 32-bit registers: UxMODE, UxSTA (status), UxBRG, UxTXREG, and UxRXREG. For example, U1MODE is the mode register for UART1. The mode register is used to configure the UART and the STA register is used to check when data is available. The BRG register is used to set the baud rate. Data is transmitted or received by writing the TXREG or reading the RXREG.

The MODE register defaults to 8 data bits, 1 stop bit, no parity, and no RTS/CTS flow control, so for most applications the programmer is only interested in bit 15, the ON bit, which enables the UART.

The STA register contains bits to enable the transmit and receive pins and to check if the transmit and receive buffers are full. After setting the ON bit of the UART, the programmer must also set the UTXEN and URXEN bits (bits 10 and 12) of the STA register to enable these two pins. UTXBF (bit 9) indicates that the transmit buffer is full. URXDA (bit 0) indicates that the receive buffer has data available. The STA register also contains bits to indicate parity and framing errors; a framing error occurs if the start or stop bits aren't found at the expected time.

The 16-bit BRG register is used to set the baud rate to a fraction of the peripheral bus clock.

$$f_{UART} = \frac{f_{peripheral\_clock}}{16 \times (\mathrm{BRG}+1)} \tag{8.4}$$

Table 8.7 lists settings of BRG for common target baud rates assuming a 20 MHz peripheral clock. It is sometimes impossible to reach exactly the target baud rate. However, as long as the frequency error is much less than 5%, the phase error between the transmitter and receiver will remain small over the duration of a 10-bit frame so data is received properly. The system will then resynchronize at the next start bit.

To transmit data, wait until STA.UTXBF is clear indicating that the transmit buffer has space available, and then write the byte to TXREG. To receive data, check STA.URXDA to see if data has arrived, and then read the byte from the RXREG.

Table 8.7 BRG settings for a 20 MHz peripheral clock

Target Baud Rate	BRG	Actual Baud Rate	Error
300	4166	300	0.0%
1200	1041	1200	0.0%
2400	520	2399	0.0%
9600	129	9615	0.2%
19200	64	19231	0.2%
38400	32	37879	−1.4%
57600	21	56818	−1.4%
115200	10	113636	−1.4%

Example 8.20 SERIAL COMMUNICATION WITH A PC

Develop a circuit and a C program for a PIC32 to communicate with a PC over a serial port at 115200 baud with 8 data bits, 1 stop bit, and no parity. The PC should be running a console program such as PuTTY[3] to read and write over the serial port. The program should ask the user to type a string. It should then tell the user what she typed.

Solution: Figure 8.43 shows a schematic of the serial link. Because few PCs still have physical serial ports, we use a Plugable USB to RS-232 DB9 Serial Adapter from `plugable.com` shown in Figure 8.44 to provide a serial connection to the PC. The adapter connects to a female DE-9 connector soldered to wires that feed a transceiver, which converts the voltages from the bipolar RS-232 levels to the PIC32 microcontroller's 3.3 V level. For this example, we chose UART2 on the PIC32. The microcontroller and PC are both Data Terminal Equipment, so the TX and RX pins must be cross-connected in the circuit. The RTS/CTS handshaking from the PIC32 is not used, and the RTS and CTS on the DE9 connector are tied together so that the PC will shake its own hand.

To configure PuTTY to work with the serial link, set *Connection type* to Serial and *Speed* to 115200. Set *Serial line* to the COM port assigned by the operating system to the Serial to USB Adapter. In Windows, this can be found in the Device Manager; for example, it might be COM3. Under the *Connection → Serial* tab, set flow control to NONE or RTS/CTS. Under the *Terminal* tab, set Local Echo to Force On to have characters appear in the terminal as you type them.

---

[3] PuTTY is available for free download at *www.putty.org*.

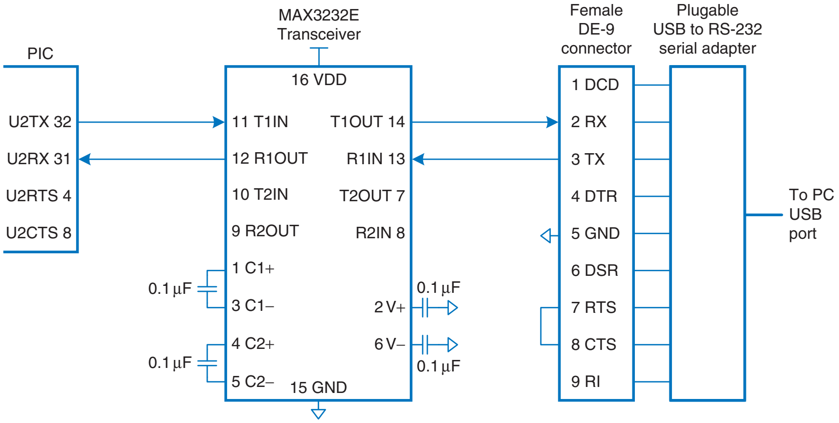

PIC
U2TX 32
U2RX 31
U2RTS 4
U2CTS 8
MAX3232E
Transceiver
16 VDD
11 T1IN
T1OUT 14
12 R1OUT
R1IN 13
10 T2IN
T2OUT 7
9 R2OUT
R2IN 8
1 C1+
3 C1–
2 V+
4 C2+
6 V–
5 C2–
15 GND
0.1 μF
0.1 μF
0.1 μF
0.1 μF
Female
DE-9
connector
1 DCD
2 RX
3 TX
4 DTR
5 GND
6 DSR
7 RTS
8 CTS
9 RI
Plugable
USB to RS-232
serial adapter
To PC
USB
port

Figure 8.43 **PIC32 to PC serial link**

Figure 8.44 **Plugable USB to RS-232 DB9 Serial Adapter** (© 2012 Plugable Technologies; reprinted with permission)

The code is listed below. The Enter key in the terminal program corresponds to a carriage return character called `'\r'` in C with an ASCII code of 0x0D. To advance to the beginning of the next line when printing, send both the `'\n'` and `'\r'` (new line and carriage return) characters.[4]

The functions to initialize, read, and write the serial port make up a simple device driver. The device driver offers a level of abstraction and modularity between the programmer and the hardware, so that a programmer using the device driver doesn't have to understand the UART's register set. It also simplifies moving the code to a different microcontroller; the device driver must be rewritten but the code that calls it can remain the same.

The `main` function demonstrates printing to the console and reading from the console using the `putstrserial` and `getstrserial` functions. It also demonstrates using `printf`, from `stdio.h`, which automatically prints through UART2. Unfortunately, the PIC32 libraries do not presently support `scanf` over the UART in a graceful way, but `getstrserial` is sufficient.

```
#include <P32xxxx.h>
#include <stdio.h>

void inituart(void) {
  U2STAbits.UTXEN = 1;        // enable transmit pin
  U2STAbits.URXEN = 1;        // enable receive pin
  U2BRG = 10;                 // set baud rate to 115.2k
  U2MODEbits.ON = 1;          // enable UART
}

char getcharserial(void) {
  while (!U2STAbits.URXDA);   // wait until data available
```

---

[4] PuTTY prints correctly even if the `\r` is omitted.

```
  return U2RXREG;                // return character received from
                                 // serial port
}

void getstrserial(char *str) {
  int i = 0;
  do {                           // read an entire string until detecting
    str[i] = getcharserial();    // carriage return
  } while (str[i++] != '\r');    // look for carriage return
  str[i–1] = 0;                  // null-terminate the string
}

void putcharserial(char c) {
  while (U2STAbits.UTXBF);       // wait until transmit buffer empty
  U2TXREG = c;                   // transmit character over serial port
}

void putstrserial(char *str) {
  int i = 0;

  while (str[i] != 0) {          // iterate over string
    putcharserial(str[i++]);     // send each character
  }
}

int main(void) {
  char str[80];

  inituart();
  while(1) {
    putstrserial("Please type something: ");
    getstrserial(str);
    printf("\n\rYou typed: %s\n\r", str);
  }
}
```

Communicating with the serial port from a C program on a PC is a bit of a hassle because serial port driver libraries are not standardized across operating systems. Other programming environments such as Python, Matlab, or LabVIEW make serial communication painless.

8.6.4 Timers

Embedded systems commonly need to measure time. For example, a microwave oven needs a timer to keep track of the time of day and another to measure how long to cook. It might use yet another to generate pulses to the motor spinning the platter, and a fourth to control the power setting by only activating the microwave's energy for a fraction of every second.

The PIC32 has five 16-bit timers on board. Timer1 is called a Type A timer that can accept an asynchronous external clock source, such as a 32 KHz watch crystal. Timers 2/3 and 4/5 are Type B timers. They run synchronously off the peripheral clock and can be paired (e.g., 2 with 3) to form 32-bit timers for measuring long periods of time.

Table 8.8 **Prescalars for Type A timers**

TCKPS[1:0]	Prescale
00	1:1
01	8:1
10	64:1
11	256:1

Table 8.9 **Prescalars for Type B timers**

TCKPS[2:0]	Prescale
000	1:1
001	2:1
010	4:1
011	8:1
100	16:1
101	32:1
110	64:1
111	256:1

Each timer is associated with three 16-bit registers: TxCON, TMRx, and PRx. For example, T1CON is the control register for Timer 1. CON is the control register. TMR contains the current time count. PR is the period register. When a timer reaches the specified period, it rolls back to 0 and sets the TxIF bit in the IFS0 interrupt flag register. A program can poll this bit to detect overflow. Alternatively, it can generate an interrupt.

By default, each timer acts as a 16-bit counter accumulating ticks of the internal peripheral clock (20 MHz in our example). Bit 15 of the CON register, called the ON bit, starts the timer counting. The TCKPS bits of the CON register specify the *prescalar*, as given in Tables 8.8 and 8.9 for Type A and Type B counters. Prescaling by *k*:1 causes the timer to only count once every *k* ticks; this can be useful to generate longer time intervals, especially when the peripheral clock is running fast. The other CON register bits are slightly different for Type A and Type B counters; see the datasheet for details.

Example 8.21 DELAY GENERATION

Write two functions that create delays of a specified number of microseconds and milliseconds using Timer1. Assume that the peripheral clock is running at 20 MHz.

Solution: Each microsecond is 20 peripheral clock cycles. We empirically observe with an oscilloscope that the `delaymicros` function has an overhead of approximately 6 µs for the function call and timer initialization. Therefore, we set PR to 20 × (`micros` – 6). Thus, the function will be inaccurate for durations less than 6 µs. The check at the start prevents overflow of the 16-bit PR.

The `delaymillis` function repeatedly invokes `delaymicros(1000)` to create an appropriate number of 1 ms delays.

```
#include <P32xxxx.h>

void delaymicros(int micros) {
  if (micros > 1000) {             // avoid timer overflow
    delaymicros(1000);
    delaymicros(micros-1000);
  }
  else if (micros > 6){
    TMR1 = 0;                      // reset timer to 0
    T1CONbits.ON = 1;              // turn timer on
    PR1 = (micros-6)*20;           // 20 clocks per microsecond.
                                   // Function has overhead of ~6 us
    IFS0bits.T1IF = 0;             // clear overflow flag
    while (!IFS0bits.T1IF);        // wait until overflow flag is set
  }
}
```

```
void delaymillis(int millis) {
  while (millis--) delaymicros(1000);  // repeatedly delay 1 ms until done
}
```

Another handy timer feature is *gated time accumulation*, in which the timer only counts while an external pin is high. This allows the timer to measure the duration of an external pulse. It is enabled using the CON register.

8.6.5 Interrupts

Timers are often used in conjunction with interrupts so that a program may go about its usual business and then periodically handle a task when the timer generates an interrupt. Section 6.7.2 described MIPS interrupts from the architectural point of view. This section explores how to use them in a PIC32.

Interrupt requests occur when a hardware event occurs, such as a timer overflowing, a character being received over a UART, or certain GPIO pins toggling. Each type of interrupt request sets a specific bit in the Interrupt Flag Status (IFS) registers. The processor then checks the corresponding bit in the Interrupt Enable Control (IEC) registers. If the bit is set, the microcontroller should respond to the interrupt request by invoking an *interrupt service routine* (ISR). The ISR is a function with `void` arguments that handles the interrupt and clears the bit of the IFS before returning. The PIC32 interrupt system supports single and multi-vector modes. In single-vector mode, all interrupts invoke the same ISR, which must examine the CAUSE register to determine the reason for the interrupt (if multiple types of interrupts may occur) and handle it accordingly. In multi-vector mode, each type of interrupt calls a different ISR. The MVEC bit in the INTCON register determines the mode. In any case, the MIPS interrupt system must be enabled with the `ei` instruction before it will accept any interrupts.

The PIC32 also allows each interrupt source to have a configurable *priority* and *subpriority*. The priority is in the range of 0–7, with 7 being highest. A higher priority interrupt will preempt an interrupt presently being handled. For example, suppose a UART interrupt is set to priority 5 and a timer interrupt is set to priority 7. If the program is executing its normal flow and a character appears on the UART, an interrupt will occur and the microcontroller can read the data from the UART and handle it. If a timer overflows while the UART ISR is active, the ISR will itself be interrupted so that the microcontroller can immediately handle the timer overflow. When it is done, it will return to finish the UART interrupt before returning to the main program. On the other hand, if the timer interrupt had priority 2, the UART ISR would complete first, then the timer ISR would be invoked, and finally the microcontroller would return to the main program.

The subpriority is in the range of 0–3. If two events with the same priority are simultaneously pending, the one with the higher subpriority will be handled first. However, the subpriority will not cause a new interrupt to preempt an interrupt of the same priority presently being serviced. The priority and subpriority of each event is configured with the IPC registers.

Each interrupt source has a vector number in the range of 0–63. For example, the Timer1 overflow interrupt is vector 4, the UART2 RX interrupt is vector 32, and the INT0 external interrupt triggered by a change on pin RD0 is vector 3. The fields of the IFS, IEC, and IPC registers corresponding to that vector number are specified in the PIC32 datasheet.

The ISR function declaration is tagged by two special `__attribute__` directives indicating the priority level and vector number. The compiler uses these attributes to associate the ISR with the appropriate interrupt request. The *Microchip MPLAB® C Compiler For PIC32 MCUs User's Guide* has more information about writing interrupt service routines.

Example 8.22 PERIODIC INTERRUPTS

Write a program that blinks an LED at 1 Hz using interrupts.

Solution: We will set Timer1 to overflow every 0.5 seconds and toggle the LED between ON and OFF in the interrupt handler.

The code below demonstrates the multi-vector mode of operation, even though only the Timer1 interrupt is actually enabled. The `blinkISR` function has attributes indicating that it has priority level 7 (IPL7) and is for vector 4 (the Timer 1 overflow vector). The ISR toggles the LED and clears the Timer1 Interrupt Flag (T1IF) bit of IFS0 before returning.

The `initTimer1Interrupt` function sets up the timer to a period of ½ second using a 256:1 prescalar and a count of 39063 ticks. It enables multi-vector mode. The priority and subpriority are specified in bits 4:2 and 1:0 of the IPC1 register, respectively. The Timer 1 interrupt flag (T1IF, bit 4 of IFS0) is cleared and the Timer1 interrupt enable (T1IE, bit 4 of IEC0) is set to accept interrupts from Timer 1. Finally, the `asm` directive is used to generate the `ei` instruction to enable the interrupt system.

The `main` function just waits in a `while` loop after initializing the timer interrupt. However, it could do something more interesting such as play a game with the user, and yet the interrupt will still ensure that the LED blinks at the correct rate.

```
#include <p32xxxx.h>

// The Timer 1 interrupt is Vector 4, using enable bit IEC0<4>
// and flag bit IFS0<4>, priority IPC1<4:2>, subpriority IPC1<1:0>
```

```
void __attribute__((interrupt(IPL7))) __attribute__((vector(4)))
blinkISR(void) {
  PORTDbits.RD0 = !PORTDbits.RD0;  // toggle the LED
  IFS0bits.T1IF = 0;               // clear the interrupt flag
  return;
}

void initTimer1Interrupt(void) {
  T1CONbits.ON = 0;       // turn timer off
  TMR1 = 0;               // reset timer to 0

  T1CONbits.TCKPS = 3;    // 1:256 prescale: 20 MHz / 256 = 78.125 KHz
  PR1 = 39063;            // toggle every half-second (one second period)

  INTCONbits.MVEC = 1;    // enable multi-vector mode – we're using vector 4
  IPC1 = 0x7 << 2 | 0x3;  // priority 7, subpriority 3
  IFS0bits.T1IF = 0;      // clear the Timer 1 interrupt flag
  IEC0bits.T1IE = 1;      // enable the Timer 1 interrupt
  asm volatile("ei");     // enable interrupts on the micro-controller
  T1CONbits.ON = 1;       // turn timer on
}

int main(void) {
  TRISD = 0;              // set up PORTD to drive LEDs
  PORTD = 0;

  initTimer1Interrupt();

  while(1);               // just wait, or do something useful here
}
```

8.6.6 Analog I/O

The real world is an analog place. Many embedded systems need analog inputs and outputs to interface with the world. They use *analog-to-digital-converters* (ADCs) to quantize analog signals into digital values, and *digital-to-analog-converters* (DACs) to do the reverse. Figure 8.45 shows symbols for these components. Such converters are characterized by their resolution, dynamic range, sampling rate, and accuracy. For example, an ADC might have $N = 12$-bit resolution over a range V_{ref-} to V_{ref+} of 0–5 V with a sampling rate of $f_s = 44$ KHz and an accuracy of ± 3 least significant bits (lsbs). Sampling rates are also listed in samples per second (sps), where 1 sps = 1 Hz. The relationship between the analog input voltage $V_{in}(t)$ and the digital sample $X[n]$ is

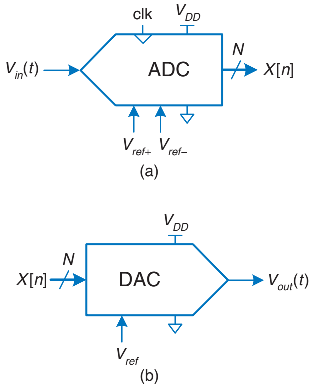


Figure 8.45 **ADC and DAC symbols**

$$\begin{aligned} X[n] &= 2^N \frac{V_{in}(t) - V_{ref-}}{V_{ref+} - V_{ref-}} \\ n &= \frac{t}{f_s} \end{aligned} \tag{8.5}$$

For example, an input voltage of 2.5 V (half of full scale) would correspond to an output of $100000000000_2 = 800_{16}$, with an uncertainty of up to 3 lsbs.

Similarly, a DAC might have $N = 16$-bit resolution over a full-scale output range of $V_{ref} = 2.56$ V. It produces an output of

$$V_{out}(t) = \frac{X[n]}{2^N} V_{ref} \tag{8.6}$$

Many microcontrollers have built-in ADCs of moderate performance. For higher performance (e.g., 16-bit resolution or sampling rates in excess of 1 MHz), it is often necessary to use a separate ADC connected to the microcontroller. Fewer microcontrollers have built-in DACs, so separate chips may also be used. However, microcontrollers often simulate analog outputs using a technique called *pulse-width modulation* (PWM). This section describes such analog I/O in the context of the PIC32 microcontroller.

A/D Conversion

The PIC32 has a 10-bit ADC with a maximum speed of 1 million samples/sec (Msps). The ADC can connect to any of 16 analog input pins via an analog multiplexer. The analog inputs are called AN0-15 and share their pins with digital I/O port RB. By default, V_{ref^+} is the analog V_{DD} pin and V_{ref^-} is the analog GND pin; in our system, these are 3.3 and 0 V, respectively. The programmer must initialize the ADC, specify which pin to sample, wait long enough to sample the voltage, start the conversion, wait until it finishes, and read the result.

The ADC is highly configurable, with the ability to automatically scan through multiple analog inputs at programmable intervals and to generate interrupts upon completion. This section simply describes how to read a single analog input pin; see the *PIC32 Family Reference Manual* for details on the other features.

The ADC is controlled by a host of registers: AD1CON1-3, AD1CHS, AD1PCFG, AD1CSSL, and ADC1BUF0-F. AD1CON1 is the primary control register. It has an ON bit to enable the ADC, a SAMP bit to control when sampling and conversion takes place, and a DONE bit to indicate conversion complete. AD1CON3 has ADCS[7:0] bits that control the speed of the A/D conversion. AD1CHS is the channel selection register specifying the analog input to sample. AD1PCFG is the pin configuration register. When a bit is 0, the corresponding pin acts as an analog input. When it is a 1, the pin acts as a digital input. ADC1BUF0 holds the 10-bit conversion result. The other registers are not required in our simple example.

Note that the ADC has a confusing use of the terms *sampling rate* and *sampling time*. The sampling time, also called the *acquisition time*, is the amount of time necessary for the input to settle before conversion takes place. The sampling rate is the number of samples taken per second. It is at most 1/(sampling time + 12 T_{AD}).

The ADC uses a successive approximation register that produces one bit of the result on each ADC clock cycle. Two additional cycles are required, for a total of 12 ADC clocks/conversion. The ADC clock period T_{AD} must be at least 65 ns for correct operation. It is set as a multiple of the peripheral clock period T_{PB} using the ADCS bits according to the relation:

$$T_{AD} = 2T_{PB}(\text{ADCS} + 1) \tag{8.7}$$

Hence, for peripheral clocks up to 30 MHz, the ADCS bits can be left at their default value of 0.

The sampling time is the amount of time necessary for a new input to stabilize before its conversion can start. As long as the resistance of the source being sampled is less than 5 KΩ, the sampling time can be as small as 132 ns, which is only a small number of clock cycles.

Example 8.23 ANALOG INPUT

Write a program to read the analog value on the AN11 pin.

Solution: The `initadc` function initializes the ADC and selects the specified channel. It leaves the ADC in sampling mode. The `readadc` function ends sampling and starts the conversion. It waits until the conversion is done, then resumes sampling and returns the conversion result.

```
#include <P32xxxx.h>

void initadc(int channel) {
  AD1CHSbits.CH0SA = channel;  // select which channel to sample
  AD1PCFGCLR = 1 << channel;   // configure pin for this channel to
                               // analog input
  AD1CON1bits.ON = 1;          // turn ADC on
  AD1CON1bits.SAMP = 1;        // begin sampling
  AD1CON1bits.DONE = 0;        // clear DONE flag
}

int readadc(void) {
  AD1CON1bits.SAMP = 0;        // end sampling, start conversion
  while (!AD1CON1bits.DONE);   // wait until done converting
  AD1CON1bits.SAMP = 1;        // resume sampling to prepare for next
                               // conversion
  AD1CON1bits.DONE = 0;        // clear DONE flag
  return ADC1BUF0;             // return conversion result
}

int main(void) {
  int sample;

  initadc(11);
  sample = readadc();
}
```

D/A Conversion

The PIC32 has no built-in DAC, so this section describes D/A conversion using external DACs. It also illustrates interfacing the PIC32 to other chips over the parallel and serial ports. The same approach could be used to interface the PIC32 to a higher resolution or faster external ADC.

Some DACs accept the *N*-bit digital input on a parallel interface with *N* wires, while others accept it over a serial interface such as SPI. Some DACs require both positive and negative power supply voltages, while others operate off of a single supply. Some support a flexible range of supply voltages, while others demand a specific voltage. The input logic levels should be compatible with the digital source. Some DACs produce a voltage output proportional to the digital input, while others produce a current output; an operational amplifier may be needed to convert this current to a voltage in the desired range.

In this section, we use the Analog Devices AD558 8-bit parallel DAC and the Linear Technology LTC1257 12-bit serial DAC. Both produce voltage outputs, run off a single 5-15 V power supply, use $V_{IH} = 2.4$ V such that they are compatible with 3.3 V outputs from the PIC32, come in DIP packages that make them easy to breadboard, and are easy to use. The AD558 produces an output on a scale of 0-2.56 V, consumes 75 mW, comes in a 16-pin package, and has a 1 μs settling time permitting an output rate of 1 Msample/sec. The datasheet is at `analog.com`. The LTC1257 produces an output on a scale of 0-2.048V, consumes less than 2 mW, comes in an 8-pin package, and has a 6 μs settling time. Its SPI operates at a maximum of 1.4 MHz. The datasheet is at `linear.com`. Texas Instruments is another leading ADC and DAC manufacturer.

Example 8.24 ANALOG OUTPUT WITH EXTERNAL DACs

Sketch a circuit and write the software for a simple signal generator producing sine and triangle waves using a PIC32, an AD558, and an LTC1257.

Solution: The circuit is shown in Figure 8.46. The AD558 connects to the PIC32 via the RD 8-bit parallel port. It connects *Vout Sense* and *Vout Select* to *Vout* to set the 2.56 V full-scale output range. The LTC1257 connects to the PIC32 via SPI2. Both ADCs use a 5 V power supply and have a 0.1 μF decoupling capacitor to reduce power supply noise. The active-low chip enable and load signals on the DACs indicate when to convert the next digital input. They should be held high while a new input is being loaded.

The program is shown below. `initio` initializes the parallel and serial ports and sets up a timer with a period to produce the desired output frequency. The SPI is set to 16-bit mode at 1 MHz, but the LTC1257 only cares about the last 12 bits sent. `initwavetables` precomputes an array of sample values for the sine and triangle waves. The sine wave is set to a 12-bit scale and the triangle to an 8-bit scale. There are 64 points per period of each wave; changing this value trades precision for frequency. `genwaves` cycles through the samples. For each sample, it disables the CE and LOAD signals to the DACs, sends the new sample over the parallel and serial ports, reenables the DACs, and then waits until the timer indicates that it is time for the next sample. The minimum sine and triangle wave frequency of 5 Hz is set by the 16-bit Timer1 period register, and the maximum

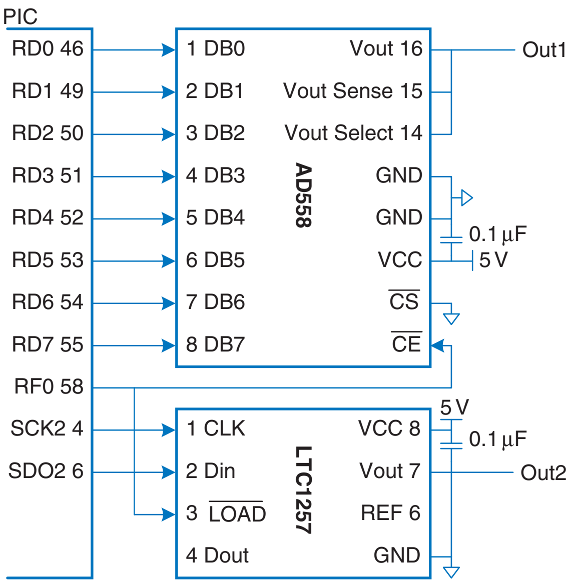

PIC
RD0 46
RD1 49
RD2 50
RD3 51
RD4 52
RD5 53
RD6 54
RD7 55
RF0 58
SCK2 4
SDO2 6
1 DB0
2 DB1
3 DB2
4 DB3
5 DB4
6 DB5
7 DB6
8 DB7
AD558
Vout 16
Vout Sense 15
Vout Select 14
GND
GND
VCC
CS
CE
Out1
0.1 μF
5 V
1 CLK
2 Din
3 LOAD
4 Dout
LTC1257
VCC 8
Vout 7
REF 6
GND
5 V
0.1 μF
Out2

Figure 8.46 DAC parallel and serial interfaces to a PIC32

frequency of 605 Hz (38.7 Ksamples/sec) is set by the time to send each point in the `genwaves` function, of which the SPI transmission is a major component.

```
#include <P32xxxx.h>
#include <math.h>                // required to use the sine function

#define NUMPTS 64

int sine[NUMPTS], triangle[NUMPTS];

void initio(int freq) {          // freq can be 5-605 Hz
  TRISD = 0xFF00;                // make the bottom 8 bits of PORT D outputs

  SPI2CONbits.ON = 0;            // disable SPI to reset any previous state
  SPI2BRG = 9;                   // 1 MHz SPI clock
  SPI2CONbits.MSTEN = 1;         // enable master mode
  SPI2CONbits.CKE = 1;           // set clock-to-data timing
  SPI2CONbits.MODE16 = 1;        // activate 16-bit mode
  SPI2CONbits.ON = 1;            // turn SPI on

  TRISF = 0xFFFE;                // make RF0 an output to control load and ce
  PORTFbits.RF0 = 1;             // set RF0 = 1

  PR1 = (20e6/NUMPTS)/freq - 1;  // set period register for desired wave
                                 // frequency
  T1CONbits.ON = 1;              // turn Timer1 on
}

void initwavetables(void) {
  int i;

  for (i=0; i<NUMPTS; i++) {
    sine[i] = 2047*(sin(2*3.14159*i/NUMPTS) + 1);     // 12-bit scale
    if (i<NUMPTS/2) triangle[i] = i*511/NUMPTS;       // 8-bit scale
    else      triangle[i] = 510-i*511/NUMPTS;
  }
}

void genwaves(void) {
  int i;

  while (1) {
    for (i=0; i<NUMPTS; i++) {
      IFS0bits.T1IF = 0;           // clear timer overflow flag
      PORTFbits.RF0=1;             // disable load while inputs are changing
      SPI2BUF = sine[i];           // send current points to the DACs
      PORTD = triangle[i];
      while (SPI2STATbits.SPIBUSY); // wait until transfer completes
      PORTFbits.RF0 = 0;           // load new points into DACs
      while (!IFS0bits.T1IF);      // wait until time to send next point
    }
  }
}

int main(void) {
  initio(500);
  initwavetables();
  genwaves();
}
```

Pulse-Width Modulation

Another way for a digital system to generate an analog output is with *pulse-width modulation* (PWM), in which a periodic output is pulsed high for part of the period and low for the remainder. The duty cycle is the fraction of the period for which the pulse is high, as shown in Figure 8.47. The average value of the output is proportional to the duty cycle. For example, if the output swings between 0 and 3.3 V and has a duty cycle of 25%, the average value will be $0.25 \times 3.3 = 0.825$ V. Low-pass filtering a PWM signal eliminates the oscillation and leaves a signal with the desired average value.

The PIC32 contains five *output compare* modules, OC1-OC5, that each, in conjunction with Timer 2 or 3, can produce PWM outputs.[5] Each output compare module is associated with three 32-bit registers: OCxCON,

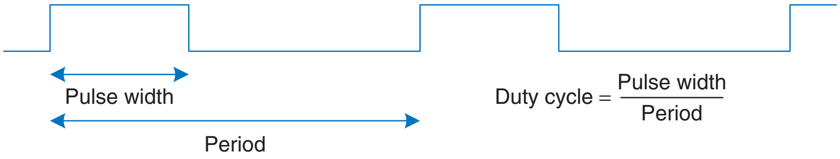


Figure 8.47 **Pulse-width modulated (PWM) signal**

---

[5] The output compare modules can also be configured to generate a single pulse based on a timer.

OCxR, and OCxRS. CON is the control register. The OCM bits of the CON register should be set to 110_2 to activate PWM mode, and the ON bit should be enabled. By default, the output compare uses Timer2 in 16-bit mode, but the OCTSEL and OC32 bits can be used to select Timer3 and/or 32-bit mode. In PWM mode, RS sets the duty cycle, the timer's period register PR sets the period, and OCxR can be ignored.

Example 8.25 ANALOG OUTPUT WITH PWM

Write a function to generate an analog output voltage using PWM and an external RC filter. The function should accept an input between 0 (for 0 V output) and 256 (for full 3.3 V output).

Solution: Use the OC1 module to produce a 78.125 KHz signal on the OC1 pin. The low-pass filter in Figure 8.48 has a corner frequency of

$$f_c = \frac{1}{2\pi RC} = 1.6\,\text{KHz}$$

to eliminate the high-speed oscillations and pass the average value.

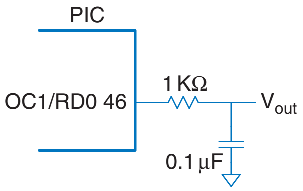


Figure 8.48 Analog output using PWM and low-pass filter

The timer should run at 20 MHz with a period of 256 ticks because 20 MHz / 256 gives the desired 78.125 KHz PWM frequency. The duty cycle input is the number of ticks for which the output should be high. If the duty cycle is 0, the output will stay low. If it is 256 or greater, the output will stay high.

The PWM code uses OC1 and Timer2. The period register is set to 255 for a period of 256 ticks. OC1RS is set to the desired duty cycle. OC1 is then configured in PWM mode and the timer and output compare module are turned ON. The program may move on to other tasks while the output compare module continues running. The OC1 pin will continuously generate the PWM signal until it is explicitly turned off.

```
#include <P32xxxx.h>

void genpwm(int dutycycle) {
  PR2 = 255;                  // set period to 255+1 ticks = 78.125 KHz
  OC1RS = dutycycle;          // set duty cycle
  OC1CONbits.OCM = 0b110;     // set output compare 1 module to PWM mode
  T2CONbits.ON = 1;           // turn on timer 2 in default mode (20 MHz,
                              // 16-bit)
  OC1CONits.ON = 1;           // turn on output compare 1 module
}
```

8.6.7 Other Microcontroller Peripherals

Microcontrollers frequently interface with other external peripherals. This section describes a variety of common examples, including character-mode liquid crystal displays (LCDs), VGA monitors, Bluetooth wireless links, and motor control. Standard communication interfaces including USB and Ethernet are described in Sections 8.7.1 and 8.7.4.

Character LCDs

A *character LCD* is a small liquid crystal display capable of showing one or a few lines of text. They are commonly used in the front panels of appliances such as cash registers, laser printers, and fax machines that need to display a limited amount of information. They are easy to interface with a microcontroller over parallel, RS-232, or SPI interfaces. Crystalfontz America sells a wide variety of character LCDs ranging from 8 columns × 1 row to 40 columns × 4 rows with choices of color, backlight, 3.3 / 5 V operation, and daylight visibility. Their LCDs can cost $20 or more in small quantities, but prices come down to under $5 in high volume.

This section gives an example of interfacing a PIC32 microcontroller to a character LCD over an 8-bit parallel interface. The interface is compatible with the industry-standard HD44780 LCD controller originally developed by Hitachi. Figure 8.49 shows a Crystalfontz CFAH2002A-TMI-JT 20 × 2 parallel LCD.

Figure 8.50 shows the LCD connected to a PIC32 over an 8-bit parallel interface. The logic operates at 5 V but is compatible with 3.3 V inputs from the PIC32. The LCD contrast is set by a second voltage produced with a potentiometer; it is usually most readable at a setting of 4.2–4.8 V. The LCD receives three control signals: RS (1 for characters, 0 for instructions), R/$\overline{W}$ (1 to read from the display, 0 to write), and E (pulsed high for at least 250 ns to enable the LCD when the next byte is ready). When the instruction is read, bit 7 returns the busy flag, indicating 1 when busy and 0 when the LCD is ready to accept another instruction. However, certain initialization steps and the clear instruction require a specified delay instead of checking the busy flag.

To initialize the LCD, the PIC32 must write a sequence of instructions to the LCD as shown below:

- Wait > 15000 µs after V_{DD} is applied
- Write 0x30 to set 8-bit mode

*** CRYSTALFONTZ ***
*** CFAH2002ATMI ***

Figure 8.49 Crystalfontz CFAH2002A-TMI 20 × 2 character LCD
(© 2012 Crystalfontz America; reprinted with permission.)

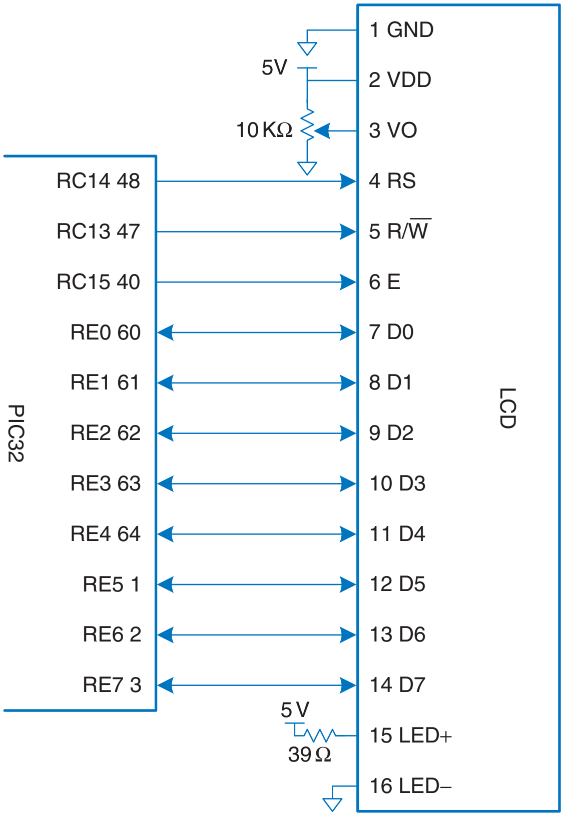

1 GND
5V
2 VDD
10 KΩ
3 VO
RC14 48
4 RS
RC13 47
5 R/$\overline{W}$
RC15 40
6 E
RE0 60
7 D0
RE1 61
8 D1
PIC32
RE2 62
9 D2
LCD
RE3 63
10 D3
RE4 64
11 D4
RE5 1
12 D5
RE6 2
13 D6
RE7 3
14 D7
5 V
15 LED+
39 Ω
16 LED–

Figure 8.50 **Parallel LCD interface**

- Wait > 4100 μs
- Write 0x30 to set 8-bit mode again
- Wait > 100 μs
- Write 0x30 to set 8-bit mode yet again
- Wait until busy flag is clear
- Write 0x3C to set 2 lines and 5 × 8 dot font
- Wait until busy flag is clear
- Write 0x08 to turn display OFF
- Wait until busy flag is clear
- Write 0x01 to clear the display
- Wait > 1530 μs

- Write 0x06 to set entry mode to increment cursor after each character
- Wait until busy flag is clear
- Write 0x0C to turn display ON with no cursor

Then, to write text to the LCD, the microcontroller can send a sequence of ASCII characters. It may also send the instructions 0x01 to clear the display or 0x02 to return to the home position in the upper left.

Example 8.26 LCD CONTROL

Write a program to write "I love LCDs" to a character display.

Solution: The following program writes "I love LCDs" to the display. It requires the `delaymicros` function from Example 8.21.

```
#include <P32xxxx.h>

typedef enum {INSTR, DATA} mode;

char lcdread(mode md) {
  char c;

  TRISE = 0xFFFF;                   // make PORTE[7:0] input
  PORTCbits.RC14 = (md == DATA);    // set instruction or data mode
  PORTCbits.RC13 = 1;               // read mode
  PORTCbits.RC15 = 1;               // pulse enable
  delaymicros(10);                  // wait for LCD to respond
  c = PORTE & 0x00FF;               // read a byte from port E
  PORTCbits.RC15 = 0;               // turn off enable
  delaymicros(10);                  // wait for LCD to respond
}

void lcdbusywait(void)
{
  char state;
  do {
    state = lcdread(INSTR);       // read instruction
  } while (state & 0x80);         // repeat until busy flag is clear
}

char lcdwrite(char val, mode md) {
  TRISE = 0xFF00;                   // make PORTE[7:0] output
  PORTCbits.RC14 = (md == DATA);    // set instruction or data mode
  PORTCbits.RC13 = 0;               // write mode
  PORTE = val;                      // value to write
  PORTCbits.RC15 = 1;               // pulse enable
  delaymicros(10);                  // wait for LCD to respond
  PORTCbits.RC15 = 0;               // turn off enable
  delaymicros(10);                  //  wait for LCD to respond
}
```

```
char lcdprintstring(char *str)
{
  while(*str != 0) {       // loop until null terminator
    lcdwrite(*str, DATA);  // print this character
    lcdbusywait();
    str++;                 // advance pointer to next character in string
  }
}

void lcdclear(void)
{
  lcdwrite(0x01, INSTR); // clear display
  delaymicros(1530);     // wait for execution
}

void initlcd(void) {
  // set LCD control pins
  TRISC = 0x1FFF;         // PORTC[15:13] are outputs, others are inputs
  PORTC = 0x0000;         // turn off all controls

  // send instructions to initialize the display
  delaymicros(15000);
  lcdwrite(0x30, INSTR); // 8-bit mode
  delaymicros(4100);
  lcdwrite(0x30, INSTR); // 8-bit mode
  delaymicros(100);
  lcdwrite(0x30, INSTR); // 8-bit mode yet again!
  lcdbusywait();
  lcdwrite(0x3C, INSTR); // set 2 lines, 5x8 font
  lcdbusywait();
  lcdwrite(0x08, INSTR); // turn display off
  lcdbusywait();
  lcdclear();
  lcdwrite(0x06, INSTR); // set entry mode to increment cursor
  lcdbusywait();
  lcdwrite(0x0C, INSTR); // turn display on with no cursor
  lcdbusywait();
}

int main(void) {
  initlcd();
  lcdprintstring("I love LCDs");
}
```

VGA Monitor

A more flexible display option is to drive a computer monitor. The *Video Graphics Array* (VGA) monitor standard was introduced in 1987 for the IBM PS/2 computers, with a 640 × 480 pixel resolution on a *cathode ray tube* (CRT) and a 15-pin connector conveying color information with analog voltages. Modern LCD monitors have higher resolution but remain backward compatible with the VGA standard.

In a cathode ray tube, an electron gun scans across the screen from left to right exciting fluorescent material to display an image. Color CRTs use three different phosphors for red, green, and blue, and three electron beams. The strength of each beam determines the intensity of each color in the pixel. At the end of each scanline, the gun must turn off for a *horizontal blanking interval* to return to the beginning of the next line. After all of the scanlines are complete, the gun must turn off again for a *vertical blanking interval* to return to the upper left corner. The process repeats about 60–75 times per second to give the visual illusion of a steady image.

In a 640 × 480 pixel VGA monitor refreshed at 59.94 Hz, the pixel clock operates at 25.175 MHz, so each pixel is 39.72 ns wide. The full screen can be viewed as 525 horizontal scanlines of 800 pixels each, but only 480 of the scanlines and 640 pixels per scan line actually convey the image, while the remainder are black. A scanline begins with a *back porch*, the blank section on the left edge of the screen. It then contains 640 pixels, followed by a blank *front porch* at the right edge of the screen and a horizontal sync (hsync) pulse to rapidly move the gun back to the left edge. Figure 8.51(a) shows the timing of each of these portions of the scanline, beginning with the active pixels. The entire scan line is 31.778 μs long. In the vertical direction, the screen starts with a back porch at the top, followed by 480 active scan lines, followed by a front porch at the bottom and a vertical sync (vsync) pulse to return to the top to start the next frame. A new frame is drawn 60 times per second. Figure 8.51(b) shows the vertical timing; note that the time units are now scan lines rather than pixel clocks. Higher resolutions use a faster pixel clock, up to 388 MHz at 2048 × 1536 @ 85 Hz. For example, 1024 × 768 @ 60 Hz can be achieved with a 65 MHz pixel clock.

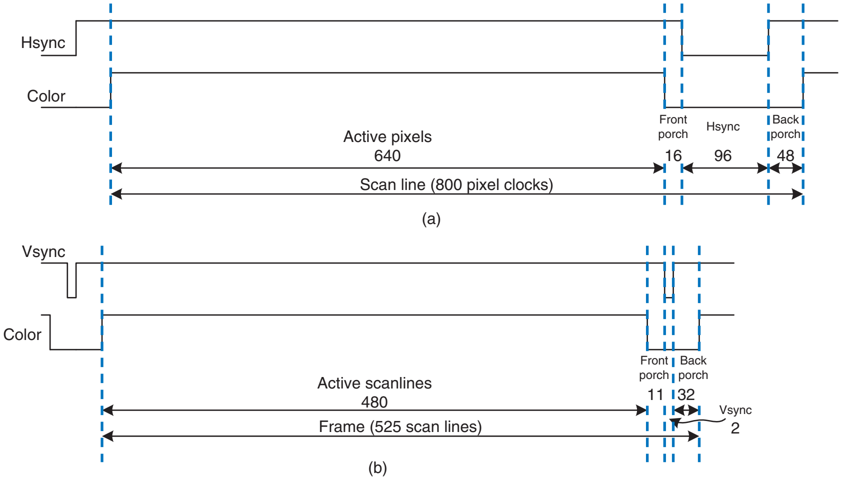


Figure 8.51 **VGA timing: (a) horizontal, (b) vertical**

The horizontal timing involves a front porch of 24 clocks, hsync pulse of 136 clocks, and back porch of 160 clocks. The vertical timing involves a front porch of 3 scan lines, vsync pulse of 6 lines, and back porch of 29 lines.

Figure 8.52 shows the pinout for a female connector coming from a video source. Pixel information is conveyed with three analog voltages for red, green, and blue. Each voltage ranges from 0–0.7 V, with more positive indicating brighter. The voltages should be 0 during the front and back porches. The cable can also provide an I^2C serial link to configure the monitor.

The video signal must be generated in real time at high speed, which is difficult on a microcontroller but easy on an FPGA. A simple black and white display could be produced by driving all three color pins with either 0 or 0.7 V using a voltage divider connected to a digital output pin. A color monitor, on the other hand, uses a *video DAC* with three separate D/A converters to independently drive the three color pins. Figure 8.53 shows

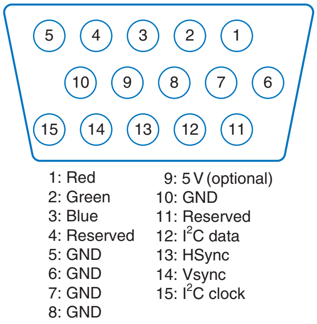

5
4
3
2
1
10
9
8
7
6
15
14
13
12
11
1: Red
2: Green
3: Blue
4: Reserved
5: GND
6: GND
7: GND
8: GND
9: 5 V (optional)
10: GND
11: Reserved
12: I2C data
13: HSync
14: Vsync
15: I2C clock

Figure 8.52 **VGA connector pinout**

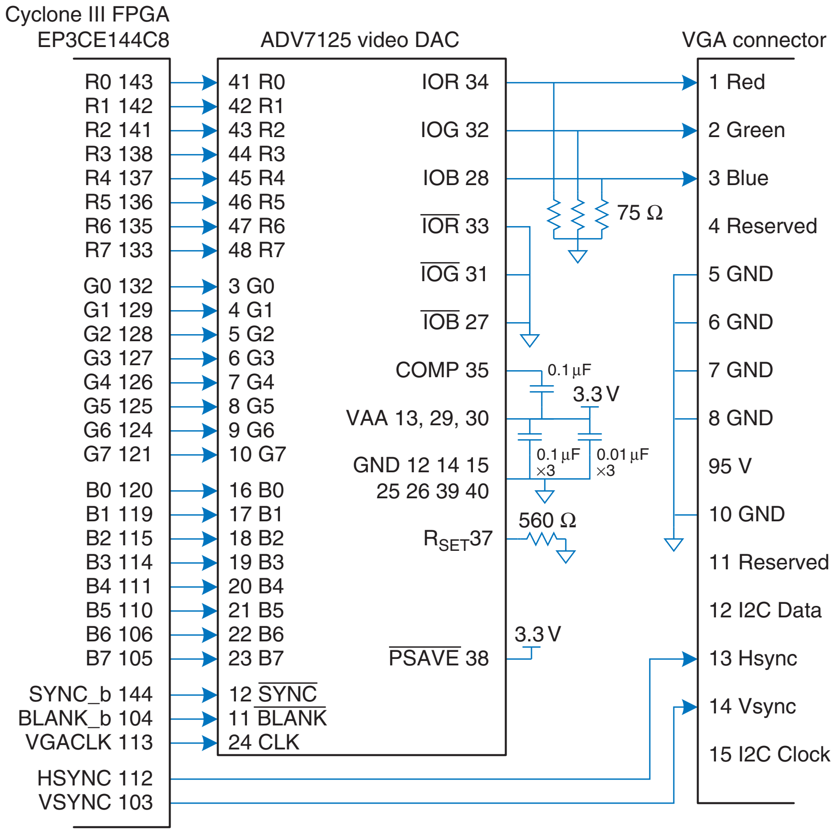

Cyclone III FPGA
EP3CE144C8
ADV7125 video DAC
VGA connector
R0 143
R1 142
R2 141
R3 138
R4 137
R5 136
R6 135
R7 133
G0 132
G1 129
G2 128
G3 127
G4 126
G5 125
G6 124
G7 121
B0 120
B1 119
B2 115
B3 114
B4 111
B5 110
B6 106
B7 105
SYNC_b 144
BLANK_b 104
VGACLK 113
HSYNC 112
VSYNC 103
41 R0
42 R1
43 R2
44 R3
45 R4
46 R5
47 R6
48 R7
3 G0
4 G1
5 G2
6 G3
7 G4
8 G5
9 G6
10 G7
16 B0
17 B1
18 B2
19 B3
20 B4
21 B5
22 B6
23 B7
12 SYNC
11 BLANK
24 CLK
IOR 34
IOG 32
IOB 28
IOR 33
IOG 31
IOB 27
COMP 35
VAA 13, 29, 30
GND 12 14 15
25 26 39 40
RSET 37
PSAVE 38
75 Ω
0.1 μF
3.3 V
0.1 μF
×3
0.01 μF
×3
560 Ω
3.3 V
1 Red
2 Green
3 Blue
4 Reserved
5 GND
6 GND
7 GND
8 GND
95 V
10 GND
11 Reserved
12 I2C Data
13 Hsync
14 Vsync
15 I2C Clock

Figure 8.53 **FPGA driving VGA cable through video DAC**

an FPGA driving a VGA monitor through an ADV7125 triple 8-bit video DAC. The DAC receives 8 bits of R, G, and B from the FPGA. It also receives a SYNC_b signal that is driven active low whenever HSYNC or VSYNC are asserted. The video DAC produces three output currents to drive the red, green, and blue analog lines, which are normally 75 Ω transmission lines parallel terminated at both the video DAC and the monitor. The R_{SET} resistor sets the scale of the output current to achieve the full range of color. The clock rate depends on the resolution and refresh rate; it may be as high as 330 MHz with a fast-grade ADV7125JSTZ330 model DAC.

Example 8.27 VGA MONITOR DISPLAY

Write HDL code to display text and a green box on a VGA monitor using the circuitry from Figure 8.53.

Solution: The code assumes a system clock frequency of 40 MHz and uses a *phase-locked loop* (PLL) on the FPGA to generate the 25.175 MHz VGA clock. PLL configuration varies among FPGAs; for the Cyclone III, the frequencies are specified with Altera's megafunction wizard. Alternatively, the VGA clock could be provided directly from a signal generator.

The VGA controller counts through the columns and rows of the screen, generating the hsync and vsync signals at the appropriate times. It also produces a blank_b signal that is asserted low to draw black when the coordinates are outside the 640 × 480 active region.

The video generator produces red, green, and blue color values based on the current (x, y) pixel location. (0, 0) represents the upper left corner. The generator draws a set of characters on the screen, along with a green rectangle. The character generator draws an 8 × 8-pixel character, giving a screen size of 80 × 60 characters. It looks up the character from a ROM, where it is encoded in binary as 6 columns by 8 rows. The other two columns are blank. The bit order is reversed by the SystemVerilog code because the leftmost column in the ROM file is the most significant bit, while it should be drawn in the least significant x-position.

Figure 8.54 shows a photograph of the VGA monitor while running this program. The rows of letters alternate red and blue. A green box overlays part of the image.

`vga.sv`

```
module vga(input  logic       clk,
           output logic       vgaclk,           // 25.175 MHz VGA clock
           output logic       hsync, vsync,
           output logic       sync_b, blank_b,  // to monitor & DAC
           output logic [7:0] r, g, b);         // to video DAC
```

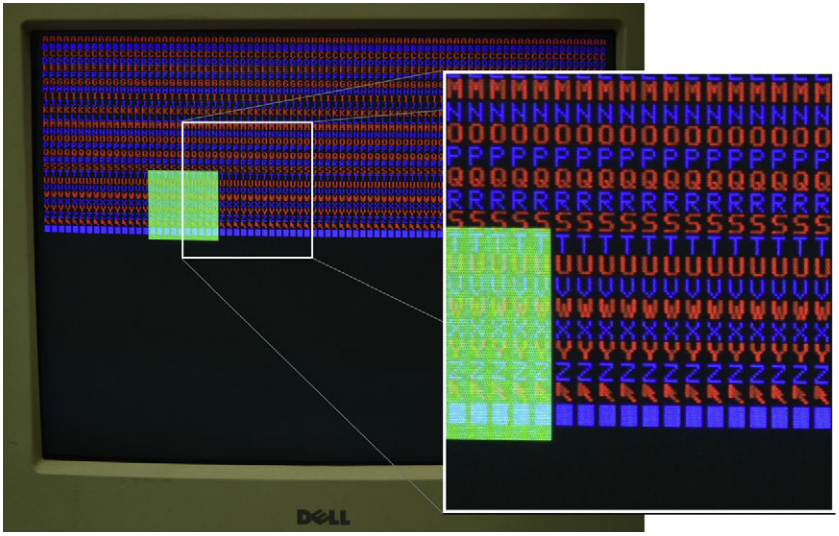


Figure 8.54 VGA output

```
  logic [9:0] x, y;

  // Use a PLL to create the 25.175 MHz VGA pixel clock
  // 25.175 MHz clk period = 39.772 ns
  // Screen is 800 clocks wide by 525 tall, but only 640 x 480 used for display
  // HSync = 1/(39.772 ns * 800) = 31.470 KHz
  // Vsync = 31.474 KHz / 525 = 59.94 Hz (~60 Hz refresh rate)
  pll   vgapll(.inclk0(clk), .c0(vgaclk));

  // generate monitor timing signals
  vgaController vgaCont(vgaclk, hsync, vsync, sync_b, blank_b, x, y);

  // user-defined module to determine pixel color
  videoGen videoGen(x, y, r, g, b);
endmodule

module vgaController #(parameter HACTIVE = 10'd640,
                                 HFP     = 10'd16,
                                 HSYN    = 10'd96,
                                 HBP     = 10'd48,
                                 HMAX    = HACTIVE + HFP + HSYN + HBP,
                                 VBP     = 10'd32,
                                 VACTIVE = 10'd480,
                                 VFP     = 10'd11,
                                 VSYN    = 10'd2,
                                 VMAX    = VACTIVE + VFP + VSYN + VBP)
                      (input  logic       vgaclk,
                       output logic       hsync, vsync, sync_b, blank_b,
                       output logic [9:0] x, y);
  // counters for horizontal and vertical positions
  always @(posedge vgaclk) begin
    x++;
    if (x == HMAX) begin
      x = 0;
      y++;
      if (y == VMAX) y = 0;
    end
  end

  // compute sync signals (active low)
  assign hsync = ~(hcnt >= HACTIVE + HFP & hcnt < HACTIVE + HFP + HSYN);
  assign vsync = ~(vcnt >= VACTIVE + VFP & vcnt < VACTIVE + VFP + VSYN);
  assign sync_b = hsync & vsync;

  // force outputs to black when outside the legal display area
  assign blank_b = (hcnt < HACTIVE) & (vcnt < VACTIVE);
endmodule

module videoGen(input logic [9:0] x, y, output logic [7:0] r, g, b);

  logic        pixel, inrect;

  // given y position, choose a character to display
  // then look up the pixel value from the character ROM
  // and display it in red or blue. Also draw a green rectangle.
  chargenrom chargenromb(y[8:3]+8'd65, x[2:0], y[2:0], pixel);
  rectgen     rectgen(x, y, 10'd120, 10'd150, 10'd200, 10'd230, inrect);
  assign {r, b} = (y[3]==0) ? {{8{pixel}},8'h00} : {8'h00,{8{pixel}}};
  assign g =        inrect    ? 8'hFF                  : 8'h00;
endmodule

module chargenrom(input  logic [7:0] ch,
                  input  logic [2:0] xoff, yoff,
                  output logic       pixel);

  logic [5:0] charrom[2047:0]; // character generator ROM
  logic [7:0] line;             // a line read from the ROM

  // initialize ROM with characters from text file
  initial
    $readmemb("charrom.txt", charrom);

  // index into ROM to find line of character
  assign line = charrom[yoff+{ch-65, 3'b000}]; // subtract 65 because A
                                               // is entry 0

  // reverse order of bits
  assign pixel = line[3'd7-xoff];
endmodule

module rectgen(input  logic [9:0] x, y, left, top, right, bot,
               output logic       inrect);

  assign inrect = (x >= left & x < right & y >= top & y < bot);
endmodule
```

charrom.txt

```
// A ASCII 65
011100
100010
100010
111110
100010
100010
100010
000000
//B ASCII 66
111100
100010
100010
111100
100010
100010
111100
000000
//C  ASCII 67
011100
100010
100000
100000
100000
100010
011100
000000
...
```

Bluetooth Wireless Communication

There are many standards now available for wireless communication, including Wi-Fi, ZigBee, and Bluetooth. The standards are elaborate and require sophisticated integrated circuits, but a growing assortment of modules abstract away the complexity and give the user a simple interface for wireless communication. One of these modules is the BlueSMiRF, which is an easy-to-use Bluetooth wireless interface that can be used instead of a serial cable.

Bluetooth is a wireless standard developed by Ericsson in 1994 for low-power, moderate speed, communication over distances of 5–100 meters, depending on the transmitter power level. It is commonly used to connect an earpiece to a cellphone or a keyboard to a computer. Unlike infrared communication links, it does not require a direct line of sight between devices.

Bluetooth operates in the 2.4 GHz unlicensed industrial-scientific-medical (ISM) band. It defines 79 radio channels spaced at 1 MHz intervals starting at 2402 MHz. It hops between these channels in a pseudo-random pattern to avoid consistent interference with other devices like wireless phones operating in the same band. As given in Table 8.10, Bluetooth transmitters are classified at one of three power levels, which dictate the

Bluetooth is named for King Harald Bluetooth of Denmark, a 10th century monarch who unified the warring Danish tribes. This wireless standard was only partially successful at unifying a host of competing wireless protocols!

Table 8.10 **Bluetooth classes**

Class	Transmitter Power (mW)	Range (m)
1	100	100
2	2.5	10
3	1	5

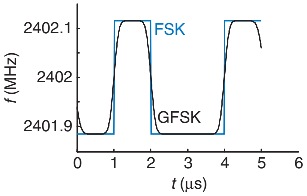


Figure 8.55 **FSK and GFSK waveforms**

range and power consumption. In the basic rate mode, it operates at 1 Mbit/sec using Gaussian frequency shift keying (FSK). In ordinary FSK, each bit is conveyed by transmitting a frequency of $f_c \pm f_d$, where f_c is the center frequency of the channel and f_d is an offset of at least 115 kHz. The abrupt transition in frequencies between bits consumes extra bandwidth. In Gaussian FSK, the change in frequency is smoothed to make better use of the spectrum. Figure 8.55 shows the frequencies being transmitted for a sequence of 0's and 1's on a 2402 MHz channel using FSK and GFSK.

A BlueSMiRF Silver module, shown in Figure 8.56, contains a Class 2 Bluetooth radio, modem, and interface circuitry on a small card with a serial interface. It communicates with another Bluetooth device such as a Bluetooth USB dongle connected to a PC. Thus, it can provide a wireless serial link between a PIC32 and a PC similar to the link from Figure 8.43 but without the cable. Figure 8.57 shows a schematic for such a link. The TX pin of the BlueSMiRF connects to the RX pin of the PIC32, and vice versa. The RTS and CTS pins are connected so that the BlueSMiRF shakes its own hand.

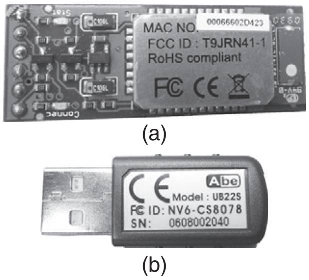


Figure 8.56 **BlueSMiRF module and USB dongle**

The BlueSMiRF defaults to 115.2k baud with 8 data bits, 1 stop bit, and no parity or flow control. It operates at 3.3 V digital logic levels, so no RS-232 transceiver is necessary to connect with another 3.3 V device.

To use the interface, plug a USB Bluetooth dongle into a PC. Power up the PIC32 and BlueSMiRF. The red STAT light will flash on the BlueSMiRF indicating that it is waiting to make a connection. Open the Bluetooth icon in the PC system tray and use the Add Bluetooth Device Wizard to pair the dongle with the BlueSMiRF. The default passkey for the BlueSMiRF is 1234. Take note of which COM port is assigned to the dongle. Then communication can proceed just as it would over a serial cable. Note that the dongle typically operates at 9600 baud and that PuTTY must be configured accordingly.

Motor Control

Another major application of microcontrollers is to drive actuators such as motors. This section describes three types of motors: DC motors, servo motors, and stepper motors. *DC motors* require a high drive current, so a powerful driver such as an *H-bridge* must be connected between the microcontroller and the motor. They also require a separate *shaft encoder*

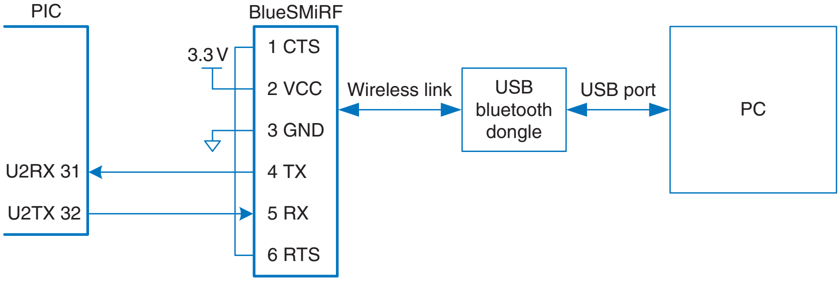


Figure 8.57 **Bluetooth PIC32 to PC link**

if the user wants to know the current position of the motor. *Servo motors* accept a pulse-width modulated signal to specify their position over a limited range of angles. They are very easy to interface, but are not as powerful and are not suited to continuous rotation. *Stepper motors* accept a sequence of pulses, each of which rotates the motor by a fixed angle called a step. They are more expensive and still need an H-bridge to drive the high current, but the position can be precisely controlled.

Motors can draw a substantial amount of current and may introduce glitches on the power supply that disturb digital logic. One way to reduce this problem is to use a different power supply or battery for the motor than for the digital logic.

DC Motors

Figure 8.58 shows the operation of a brushed DC motor. The motor is a two terminal device. It contains permanent stationary magnets called the *stator* and a rotating electromagnet called the *rotor* or *armature* connected to the shaft. The front end of the rotor connects to a split metal ring called a *commutator*. Metal brushes attached to the power lugs (input terminals) rub against the commutator, providing current to the rotor's electromagnet. This induces a magnetic field in the rotor that causes the rotor to spin to become aligned with the stator field. Once the rotor has spun part way around and approaches alignment with the stator, the brushes touch the opposite sides of the commutator, reversing the current flow and magnetic field and causing it to continue spinning indefinitely.

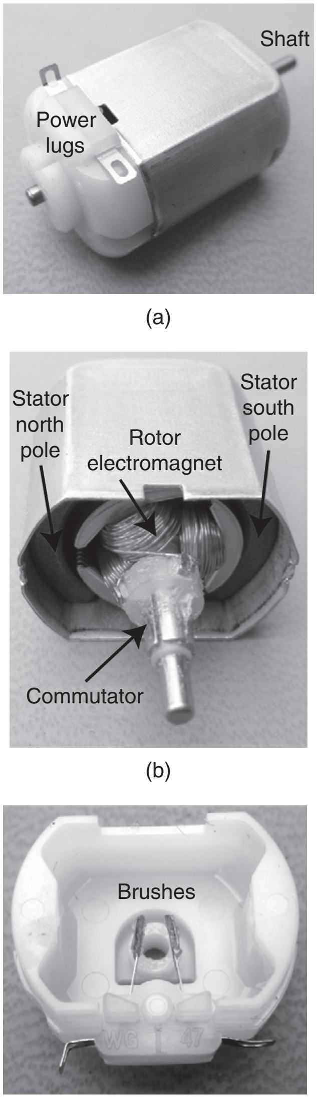


Figure 8.58 **DC motor**

DC motors tend to spin at thousands of rotations per minute (RPM) at very low torque. Most systems add a gear train to reduce the speed to a more reasonable level and increase the torque. Look for a gear train designed to mate with your motor. Pittman manufactures a wide range of high quality DC motors and accessories, while inexpensive toy motors are popular among hobbyists.

A DC motor requires substantial current and voltage to deliver significant power to a load. The current should be reversible if the motor can spin in both directions. Most microcontrollers cannot produce enough current to drive a DC motor directly. Instead, they use an H-bridge, which conceptually contains four electrically controlled switches, as shown in Figure 8.59(a). If switches A and D are closed, current flows from left to right through the motor and it spins in one direction. If B and C are closed, current flows from right to left through the motor and it spins in the other direction. If A and C or B and D are closed, the voltage across the motor is forced to 0, causing the motor to actively brake. If none of the switches are closed, the motor will coast to a stop. The switches in an H-bridge are power transistors. The H-bridge also contains some digital logic to conveniently control the switches. When the motor current changes abruptly, the inductance of the motor's electromagnet will induce a large voltage that could exceed the power supply and damage the power transistors.

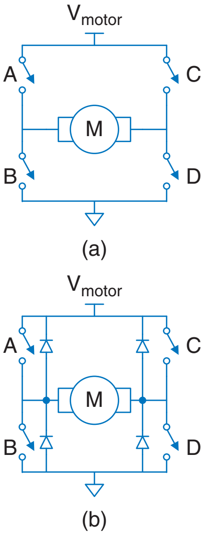

V_{motor}
A
C
M
B
D
(a)
V_{motor}
A
C
M
B
D
(b)

Figure 8.59 **H-bridge**

Therefore, many H-bridges also have protection diodes in parallel with the switches, as shown in Figure 8.59(b). If the inductive kick drives either terminal of the motor above V_{motor} or below ground, the diodes will turn ON and clamp the voltage at a safe level. H-bridges can dissipate large amounts of power, and a heat sink may be necessary to keep them cool.

Example 8.28 AUTONOMOUS VEHICLE

Design a system in which a PIC32 controls two drive motors for a robot car. Write a library of functions to initialize the motor driver and to make the car drive forward and back, turn left or right, and stop. Use PWM to control the speed of the motors.

Solution: Figure 8.60 shows a pair of DC motors controlled by a PIC32 using a Texas Instruments SN754410 dual H-bridge. The H-bridge requires a 5 V logic supply V_{CC1} and a 4.5-36 V motor supply V_{CC2}; it has $V_{IH} = 2$ V and is hence compatible with the 3.3 V I/O from the PIC32. It can deliver up to 1 A of current to each of two motors. Table 8.11 describes how the inputs to each H-bridge

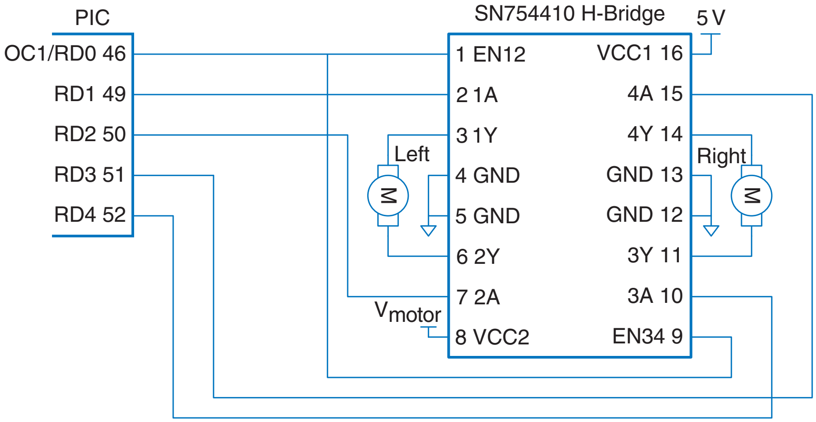

PIC
OC1/RD0 46
RD1 49
RD2 50
RD3 51
RD4 52
SN754410 H-Bridge
5 V
1 EN12
2 1A
3 1Y
4 GND
5 GND
6 2Y
7 2A
8 VCC2
VCC1 16
4A 15
4Y 14
GND 13
GND 12
3Y 11
3A 10
EN34 9
Left
Right
M
M
Vmotor

Figure 8.60 **Motor control with dual H-bridge**

Table 8.11 **H-Bridge control**

EN12	1A	2A	Motor
0	X	X	Coast
1	0	0	Brake
1	0	1	Reverse
1	1	0	Forward
1	1	1	Brake

control a motor. The microcontroller drives the enable signals with a PWM signal to control the speed of the motors. It drives the four other pins to control the direction of each motor.

The PWM is configured to work at about 781 Hz with a duty cycle ranging from 0 to 100%.

```
#include <P32xxxx.h>

void setspeed(int dutycycle) {
  OC1RS = dutycycle;                   // set duty cycle between 0 and 100
}

void setmotorleft(int dir) {           // dir of 1 = forward, 0 = backward
  PORTDbits.RD1 = dir; PORTDbits.RD2 = !dir;
}

void setmotorright(int dir) {          // dir of 1 = forward, 0 = backward
  PORTDbits.RD3 = dir; PORTDbits.RD4 = !dir;
}

void forward(void) {
  setmotorleft(1); setmotorright(1); // both motors drive forward
}

void backward(void) {
  setmotorleft(0); setmotorright(0); // both motors drive backward
}

void left(void) {
  setmotorleft(0); setmotorright(1); // left back, right forward
}

void right(void) {
  setmotorleft(1); setmotorright(0); // right back, left forward
}

void halt(void) {
  PORTDCLR = 0x001E;                   // turn both motors off by
                                       // clearing RD[4:1] to 0
}

void initmotors(void) {
  TRISD = 0xFFE0;           // RD[4:0] are outputs
  halt();                   // ensure motors aren't spinning
                            // configure PWM on OC1 (RD0)
  T2CONbits.TCKPS = 0b111; // prescale by 256 to 78.125 KHz
  PR2 = 99;                 // set period to 99+1 ticks = 781.25 Hz
  OC1RS = 0;                // start with low H-bridge enable signal
  OC1CONbits.OCM = 0b110;   // set output compare 1 module to PWM mode
  T2CONbits.ON = 1;         // turn on timer 2
  OC1CONbits.ON = 1;        // turn on PWM
}
```

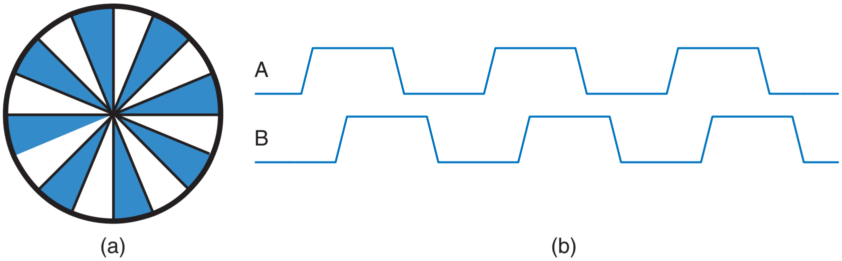


Figure 8.61 Shaft encoder (a) disk, (b) quadrature outputs

In the previous example, there is no way to measure the position of each motor. Two motors are unlikely to be exactly matched, so one is likely to turn slightly faster than the other, causing the robot to veer off course. To solve this problem, some systems add shaft encoders. Figure 8.61(a) shows a simple shaft encoder consisting of a disk with slots attached to the motor shaft. An LED is placed on one side and a light sensor is placed on the other side. The shaft encoder produces a pulse every time the gap rotates past the LED. A microcontroller can count these pulses to measure the total angle that the shaft has turned. By using two LED/sensor pairs spaced half a slot width apart, an improved shaft encoder can produce quadrature outputs shown in Figure 8.61(b) that indicate the direction the shaft is turning as well as the angle by which it has turned. Sometimes shaft encoders add another hole to indicate when the shaft is at an index position.

Servo Motor

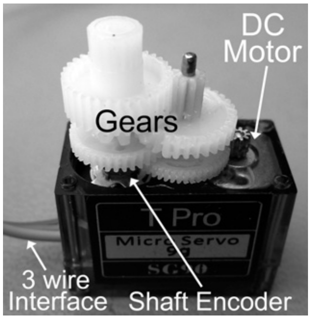


Figure 8.62 SG90 servo motor

A servo motor is a DC motor integrated with a gear train, a shaft encoder, and some control logic so that it is easier to use. They have a limited rotation, typically 180°. Figure 8.62 shows a servo with the lid removed to reveal the gears. A servo motor has a 3-pin interface with power (typically 5 V), ground, and a control input. The control input is typically a 50 Hz pulse-width modulated signal. The servo's control logic drives the shaft to a position determined by the duty cycle of the control input. The servo's shaft encoder is typically a rotary potentiometer that produces a voltage dependent on the shaft position.

In a typical servo motor with 180 degrees of rotation, a pulse width of 0.5 ms drives the shaft to 0°, 1.5 ms to 90°, and 2.5 ms to 180°. For example, Figure 8.63 shows a control signal with a 1.5 ms pulse width. Driving the servo outside its range may cause it to hit mechanical stops and be damaged. The servo's power comes from the power pin rather than the control pin, so the control can connect directly to a microcontroller without an H-bridge. Servo motors are commonly used in

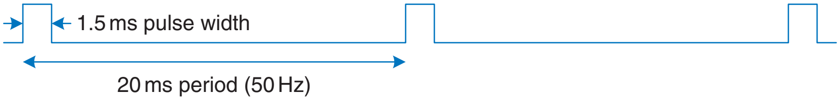


Figure 8.63 **Servo control waveform**

remote-control model airplanes and small robots because they are small, light, and convenient. Finding a motor with an adequate datasheet can be difficult. The center pin with a red wire is normally power, and the black or brown wire is normally ground.

Example 8.29 SERVO MOTOR

Design a system in which a PIC32 microcontroller drives a servo motor to a desired angle.

Solution: Figure 8.64 shows a diagram of the connection to an SG90 servo motor. The servo operates off of a 4.0–7.2 V power supply. Only a single wire is necessary to carry the PWM signal, which can be provided at 5 or 3.3 V logic levels. The code configures the PWM generation using the Output Compare 1 module and sets the appropriate duty cycle for the desired angle.

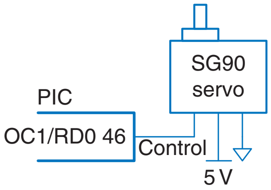


Figure 8.64 **Servo motor control**

```
#include <P32xxxx.h>

void initservo(void) {        // configure PWM on OC1 (RD0)
  T2CONbits.TCKPS = 0b111;    // prescale by 256 to 78.125 KHz
  PR2 = 1561;                 // set period to 1562 ticks = 50.016 Hz (20 ms)
  OC1RS = 117;                // set pulse width to 1.5 ms to center servo
  OC1CONbits.OCM = 0b110;     // set output compare 1 module to PWM mode
  T2CONbits.ON = 1;           // turn on timer 2
  OC1CONbits.ON = 1;          // turn on PWM
}

void setservo(int angle) {
  if (angle < 0)      angle = 0;       // angle must be in the range of
                                       // 0-180 degrees
  else if (angle > 180) angle = 180;
  OC1RS = 39+angle*156.1/180;          // set pulsewidth of 39-195 ticks
                                       // (0.5-2.5 ms) based on angle
}
```

It is also possible to convert an ordinary servo into a *continuous rotation servo* by carefully disassembling it, removing the mechanical stop, and replacing the potentiometer with a fixed voltage divider. Many websites show detailed directions for particular servos. The PWM will then control the velocity rather than position, with 1.5 ms indicating stop, 2.5 ms indicating full speed forward, and 0.5 ms indicating full speed backward. A continuous rotation servo may be more convenient and less expensive than a simple DC motor combined with an H-bridge and gear train.

Stepper Motor

A stepper motor advances in discrete steps as pulses are applied to alternate inputs. The step size is usually a few degrees, allowing precise positioning and continuous rotation. Small stepper motors generally come with two sets of coils called *phases* wired in *bipolar* or *unipolar* fashion. Bipolar motors are more powerful and less expensive for a given size but require an H-bridge driver, while unipolar motors can be driven with transistors acting as switches. This section focuses on the more efficient bipolar stepper motor.

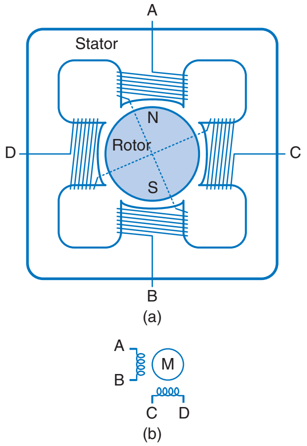


Figure 8.65 (a) Simplified bipolar stepper motor; (b) stepper motor symbol

Figure 8.65(a) shows a simplified two-phase bipolar motor with a 90° step size. The rotor is a permanent magnet with one north and one south pole. The stator is an electromagnet with two pairs of coils comprising the two phases. Two-phase bipolar motors thus have four terminals. Figure 8.65(b) shows a symbol for the stepper motor modeling the two coils as inductors.

Figure 8.66 shows three common drive sequences for a two phase bipolar motor. Figure 8.66(a) illustrates *wave drive*, in which the coils are energized in the sequence AB – CD – BA – DC. Note that BA means that the winding AB is energized with current flowing in the opposite direction; this is the origin of the name *bipolar*. The rotor turns by 90 degrees at each step. Figure 8.66(b) illustrates *two-phase-on drive*, following the pattern (AB, CD) – (BA, CD) – (BA, DC) – (AB, DC). (AB, CD) indicates that both coils AB and CD are energized simultaneously. The rotor again turns by 90 degrees at each step, but aligns itself halfway between the two pole positions. This gives the highest torque operation because both coils are delivering power at once. Figure 8.66(c) illustrates *half-step drive*, following the pattern (AB, CD) – CD – (BA, CD) – BA – (BA, DC) – DC – (AB, DC) – AB. The rotor turns by 45 degrees at each half-step. The rate at which the pattern advances determines the speed of the motor. To reverse the motor direction, the same drive sequences are applied in the opposite order.

In a real motor, the rotor has many poles to make the angle between steps much smaller. For example, Figure 8.67 shows an AIRPAX LB82773-M1 bipolar stepper motor with a 7.5 degree step size. The motor operates off 5 V and draws 0.8 A through each coil.

The torque in the motor is proportional to the coil current. This current is determined by the voltage applied and by the inductance L and resistance R of the coil. The simplest mode of operation is called *direct voltage drive* or *L/R drive*, in which the voltage V is directly

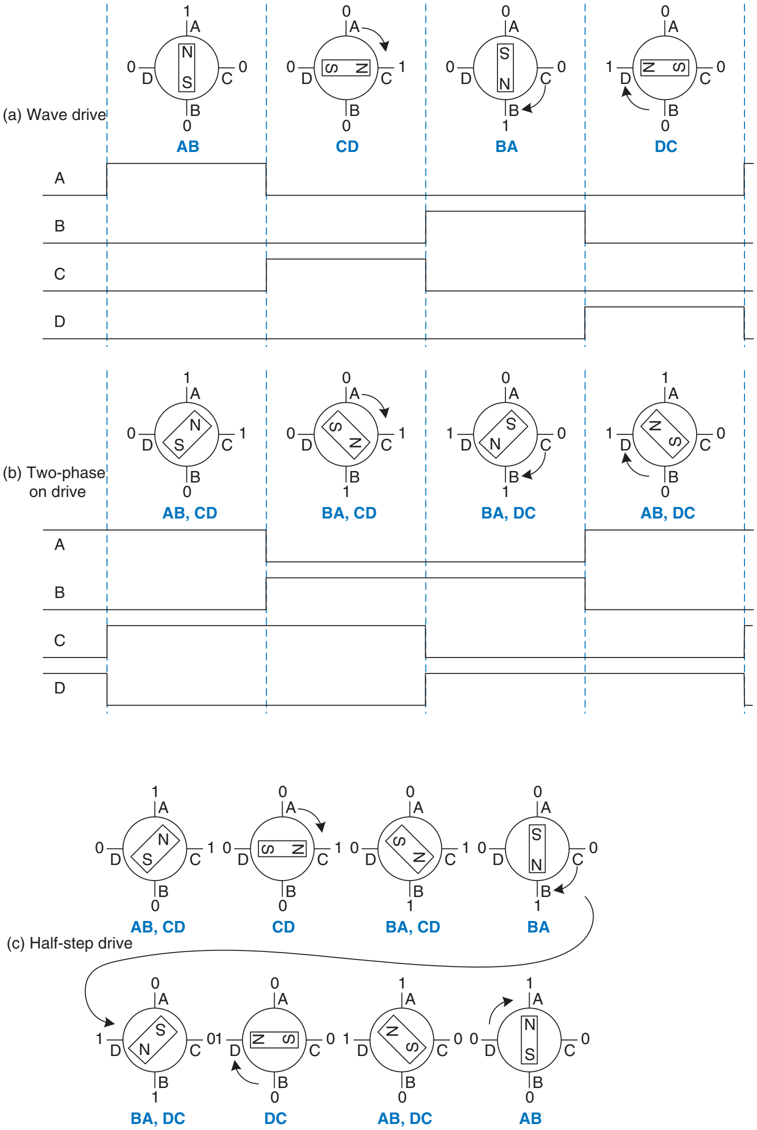

(a) Wave drive
AB
CD
BA
DC
A
B
C
D
(b) Two-phase on drive
AB, CD
BA, CD
BA, DC
AB, DC
(c) Half-step drive
AB, CD
CD
BA, CD
BA
BA, DC
DC
AB, DC
AB

Figure 8.66 Bipolar motor drive

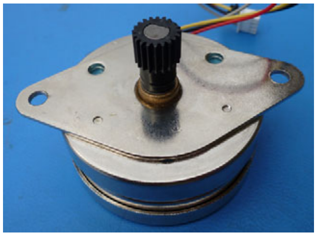


Figure 8.67 AIRPAX LB82773-M1 bipolar stepper motor

applied to the coil. The current ramps up to $I = V/R$ with a time constant set by L/R, as shown in Figure 8.68(a). This works well for slow speed operation. However, at higher speed, the current doesn't have enough time to ramp up to the full level, as shown in Figure 8.68(b), and the torque drops off.

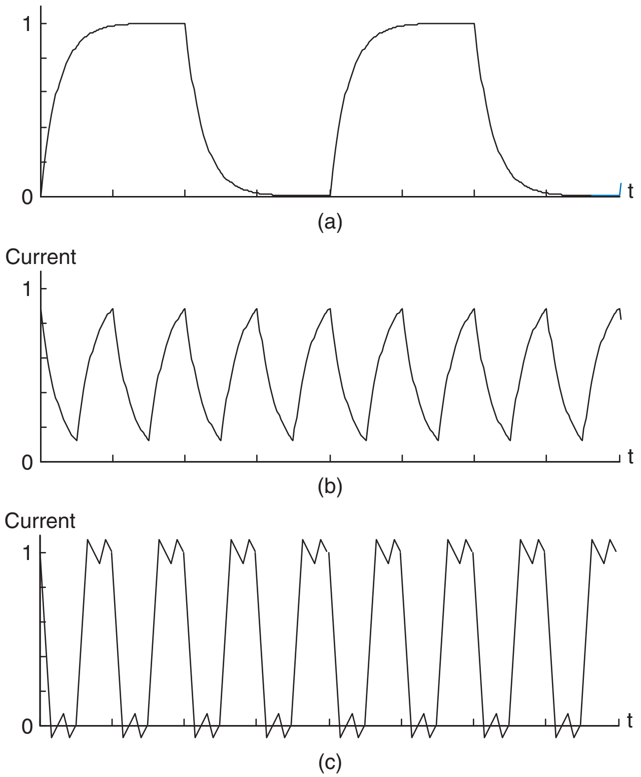


Figure 8.68 **Bipolar stepper motor direct drive current: (a) slow rotation, (b) fast rotation, (c) fast rotation with chopper drive**

A more efficient way to drive a stepper motor is by pulse-width modulating a higher voltage. The high voltage causes the current to ramp up to full current more rapidly, then it is turned off (PWM) to avoid overloading the motor. The voltage is then modulated or *chopped* to maintain the current near the desired level. This is called *chopper constant current drive* and is shown in Figure 8.68(c). The controller uses a small resistor in series with the motor to sense the current being applied by measuring the voltage drop, and applies an enable signal to the H-bridge to turn off the drive when the current reaches the desired level. In principle, a microcontroller could generate the right waveforms, but it is easier to use a stepper motor controller. The L297 controller from ST Microelectronics is a convenient choice, especially when coupled with the L298 dual H-bridge with current sensing pins and a 2 A peak power capability. Unfortunately, the L298 is not available in a DIP package so it is harder to breadboard. ST's application notes AN460 and AN470 are valuable references for stepper motor designers.

Example 8.30 BIPOLAR STEPPER MOTOR DIRECT WAVE DRIVE

Design a system in which a PIC32 microcontroller drives an AIRPAX bipolar stepper motor at a specified speed and direction using direct drive.

Solution: Figure 8.69 shows a bipolar stepper motor being driven directly by an H-bridge controlled by a PIC32.

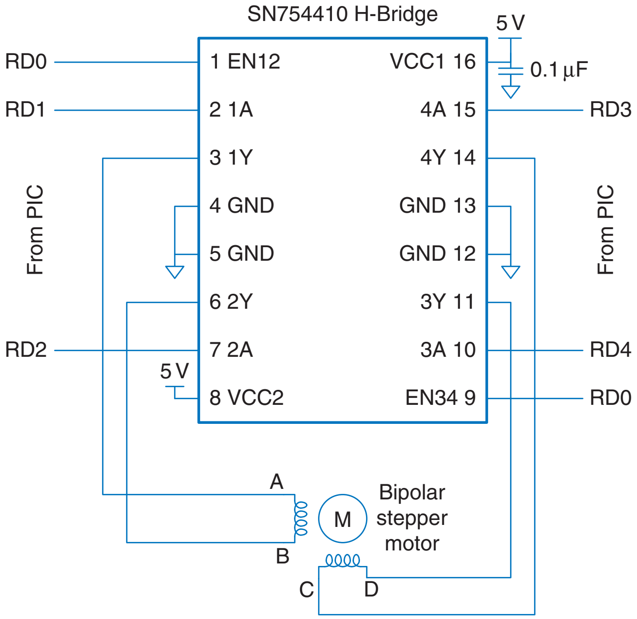

SN754410 H-Bridge
5 V
0.1 μF
RD0
RD1
1 EN12
2 1A
3 1Y
4 GND
5 GND
6 2Y
7 2A
8 VCC2
VCC1 16
4A 15
4Y 14
GND 13
GND 12
3Y 11
3A 10
EN34 9
RD3
RD4
RD0
From PIC
From PIC
RD2
5 V
A
B
C
D
M
Bipolar stepper motor

Figure 8.69 **Bipolar stepper motor direct drive with H-bridge**

The `spinstepper` function initializes the sequence array with the patterns to apply to RD[4:0] to follow the direct drive sequence. It applies the next pattern in the sequence, then waits a sufficient time to rotate at the desired revolutions per minute (RPM). Using a 20 MHz clock and a 7.5° step size with a 16-bit timer and 256:1 prescalar, the feasible range of speeds is 2–230 rpm, with the bottom end limited by the timer resolution and the top end limited by the power of the LB82773-M1 motor.

```
#include <P32xxxx.h>

#define STEPSIZE 7.5 // size of step, in degrees

int curstepstate; // keep track of current state of stepper motor in sequence

void initstepper(void) {
   TRISD = 0xFFE0;                                  // RD[4:0] are outputs
   curstepstate = 0;
```

```
    T1CONbits.ON = 0;                                          // turn Timer1 off
    T1CONbits.TCKPS = 3;                                       // prescale by 256 to run slower
}

void spinstepper(int dir, int steps, float rpm) {              // dir = 0 for forward, 1 = reverse
{
  int sequence[4] = {0b00011, 0b01001, 0b00101, 0b10001};      // wave drive sequence
  int step;

  PR1 = (int)(20.0e6/(256*(360.0/STEPSIZE)*(rpm/60.0)));       // time/step w/ 20 MHz peripheral clock
  TMR1 = 0;
  T1CONbits.ON = 1;                                            // turn Timer1 on
  for (step = 0; step < steps; step++) {                       // take specified number of steps
    PORTD = sequence[curstepstate];                            // apply current step control
    if (dir == 0) curstepstate = (curstepstate + 1) % 4;       // determine next state forward
    else          curstepstate = (curstepstate + 3) % 4;       // determine next state backward
    while (!IFS0bits.T1IF);                                    // wait for timer to overflow
    IFS0bits.T1IF = 0;                                         // clear overflow flag
  }
  T1CONbits.ON = 0;                                            // Timer1 off to save power when done
}
```

8.7 PC I/O SYSTEMS

Personal computers (PCs) use a wide variety of I/O protocols for purposes including memory, disks, networking, internal expansion cards, and external devices. These I/O standards have evolved to offer very high performance and to make it easy for users to add devices. These attributes come at the expense of complexity in the I/O protocols. This section explores the major I/O standards used in PCs and examines some options for connecting a PC to custom digital logic or other external hardware.

Figure 8.70 shows a PC motherboard for a Core i5 or i7 processor. The processor is packaged in a *land grid array* with 1156 gold-plated pads to supply power and ground to the processor and connect the processor to memory and I/O devices. The motherboard contains the DRAM memory module slots, a wide variety of I/O device connectors, and the power supply connector, voltage regulators, and capacitors. A pair of DRAM modules are connected over a DDR3 interface. External peripherals such as keyboards or webcams are attached over USB. High-performance expansion cards such as graphics cards connect over the PCI Express x16 slot, while lower-performance cards can use PCI Express x1 or the older PCI slots. The PC connects to the network using the Ethernet jack. The hard disk connects to a SATA port. The remainder of this section gives an overview of the operation of each of these I/O standards.

One of the major advances in PC I/O standards has been the development of high-speed serial links. Until recently, most I/O was built

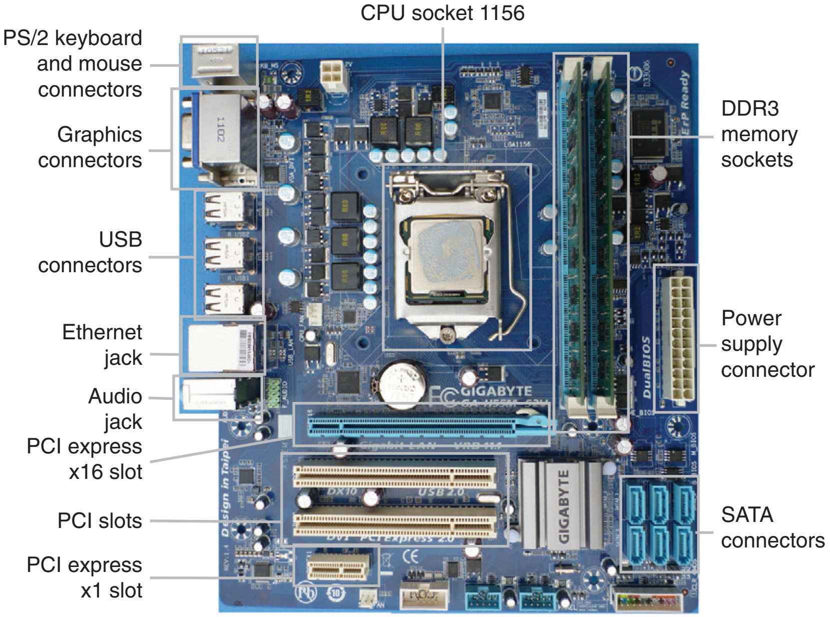

CPU socket 1156
PS/2 keyboard and mouse connectors
Graphics connectors
USB connectors
Ethernet jack
Audio jack
PCI express x16 slot
PCI slots
PCI express x1 slot
DDR3 memory sockets
Power supply connector
SATA connectors

Figure 8.70 **Gigabyte GA-H55M-S2V Motherboard**

around parallel links consisting of a wide data bus and a clock signal. As data rates increased, the difference in delay among the wires in the bus set a limit to how fast the bus could run. Moreover, busses connected to multiple devices suffer from transmission line problems such as reflections and different flight times to different loads. Noise can also corrupt the data. Point-to-point serial links eliminate many of these problems. The data is usually transmitted on a differential pair of wires. External noise that affects both wires in the pair equally is unimportant. The transmission lines are easy to properly terminate, so reflections are small (see Section A.8 on transmission lines). No explicit clock is sent; instead, the clock is recovered at the receiver by watching the timing of the data transitions. High-speed serial link design is a specialized subject, but good interfaces can run faster than 10 Gb/s over copper wires and even faster along optical fibers.

8.7.1 USB

Until the mid-1990's, adding a peripheral to a PC took some technical savvy. Adding expansion cards required opening the case, setting jumpers to the correct position, and manually installing a device driver. Adding an RS-232 device required choosing the right cable and properly configuring the baud rate, and data, parity, and stop bits. The *Universal Serial Bus* (USB), developed by Intel, IBM, Microsoft, and others, greatly simplified adding peripherals by standardizing the cables and software configuration process. Billions of USB peripherals are now sold each year.

USB 1.0 was released in 1996. It uses a simple cable with four wires: 5 V, GND, and a differential pair of wires to carry data. The cable is impossible to plug in backward or upside down. It operates at up to 12 Mb/s. A device can pull up to 500 mA from the USB port, so keyboards, mice, and other peripherals can get their power from the port rather than from batteries or a separate power cable.

USB 2.0, released in 2000, upgraded the speed to 480 Mb/s by running the differential wires much faster. With the faster link, USB became practical for attaching webcams and external hard disks. Flash memory sticks with a USB interface also replaced floppy disks as a means of transferring files between computers.

USB 3.0, released in 2008, further boosted the speed to 5 Gb/s. It uses the same shape connector, but the cable has more wires that operate at very high speed. It is better suited to connecting high-performance hard disks. At about the same time, USB added a Battery Charging Specification that boosts the power supplied over the port to speed up charging mobile devices.

The simplicity for the user comes at the expense of a much more complex hardware and software implementation. Building a USB interface from the ground up is a major undertaking. Even writing a simple device driver is moderately complex. The PIC32 comes with a built-in USB controller. However, Microchip's device driver to connect a mouse to the PIC32 (available at `microchip.com`) is more than 500 lines of code and is beyond the scope of this chapter.

8.7.2 PCI and PCI Express

The *Peripheral Component Interconnect* (PCI) bus is an expansion bus standard developed by Intel that became widespread around 1994. It was used to add expansion cards such as extra serial or USB ports, network interfaces, sound cards, modems, disk controllers, or video cards. The 32-bit parallel bus operates at 33 MHz, giving a bandwidth of 133 MB/s.

The demand for PCI expansion cards has steadily declined. More standard ports such as Ethernet and SATA are now integrated into the motherboard. Many devices that once required an expansion card can now be connected over a fast USB 2.0 or 3.0 link. And video cards now require far more bandwidth than PCI can supply.

Contemporary motherboards often still have a small number of PCI slots, but fast devices like video cards are now connected via *PCI Express* (PCIe). PCI Express slots provide one or more lanes of high-speed serial links. In PCIe 3.0, each lane operates at up to 8 Gb/s. Most motherboards provide an x16 slot with 16 lanes giving a total of 16 GB/s of bandwidth to data-hungry devices such as video cards.

8.7.3 DDR3 Memory

DRAM connects to the microprocessor over a parallel bus. In 2012, the present standard is DDR3, a third generation of double-data rate memory bus operating at 1.5 V. Typical motherboards now come with two DDR3 channels so they can access two banks of memory modules simultaneously.

Figure 8.71 shows a 4 GB DDR3 dual inline memory module (DIMM). The module has 120 contacts on each side, for a total of 240 connections, including a 64-bit data bus, a 16-bit time-multiplexed address bus, control signals, and numerous power and ground pins. In 2012, DIMMs typically carry 1–16 GB of DRAM. Memory capacity has been doubling approximately every 2–3 years.

DRAM presently operates at a clock rate of 100–266 MHz. DDR3 operates the memory bus at four times the DRAM clock rate. Moreover, it transfers data on both the rising and falling edges of the clock. Hence, it sends 8 words of data for each memory clock. At 64 bits/word, this corresponds to 6.4–17 GB/s of bandwidth. For example, DDR3-1600 uses a 200 MHz memory clock and an 800 MHz I/O clock to send 1600 million words/sec, or 12800 MB/s. Hence, the modules are also called PC3-12800. Unfortunately, DRAM latency remains high, with a roughly 50 ns lag from a read request until the arrival of the first word of data.

8.7.4 Networking

Computers connect to the Internet over a network interface running the *Transmission Control Protocol and Internet Protocol* (TCP/IP). The physical connection may be an Ethernet cable or a wireless Wi-Fi link.

Ethernet is defined by the IEEE 802.3 standard. It was developed at Xerox Palo Alto Research Center (PARC) in 1974. It originally operated at 10 Mb/s (called 10 Mbit Ethernet), but now is commonly found at 100 Mbit (Mb/s) and 1 Gbit (Gb/s) running on Category 5 cables containing four twisted pairs of wires. 10 Gbit Ethernet running on fiber optic cables is increasingly popular for servers and other high-performance computing, and 100 Gbit Ethernet is emerging.

Wi-Fi is the popular name for the IEEE 802.11 wireless network standard. It operates in the 2.4 and 5 GHz unlicensed wireless bands, meaning that the user doesn't need a radio operator's license to transmit in these

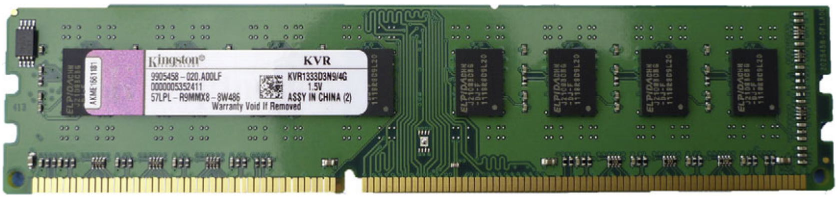


Figure 8.71 DDR3 memory module

Table 8.12 802.11 Wi-Fi Protocols

Protocol	Release	Frequency Band (GHz)	Data Rate (Mb/s)	Range (m)
802.11b	1999	2.4	5.5–11	35
802.11g	2003	2.4	6–54	38
802.11n	2009	2.4/5	7.2–150	70

bands at low power. Table 8.12 summarizes the capabilities of three generations of Wi-Fi; the emerging 802.11ac standard promises to push wireless data rates beyond 1 Gb/s. The increasing performance comes from advancing modulation and signal processing, multiple antennas, and wider signal bandwidths.

8.7.5 SATA

Internal hard disks require a fast interface to a PC. In 1986, Western Digital introduced the *Integrated Drive Electronics* (IDE) interface, which evolved into the *AT Attachment* (ATA) standard. The standard uses a bulky 40 or 80-wire ribbon cable with a maximum length of 18” to send data at 16–133 MB/s.

ATA has been supplanted by Serial ATA (SATA), which uses high-speed serial links to run at 1.5, 3, or 6 Gb/s over a more convenient 7-conductor cable shown in Figure 8.72. The fastest solid-state drives in 2012 approach 500 MB/s of bandwidth, taking full advantage of SATA.

A related standard is Serial Attached SCSI (SAS), an evolution of the parallel SCSI (Small Computer System Interface) interface. SAS offers performance comparable to SATA and supports longer cables; it is common in server computers.

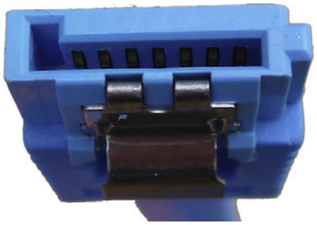


Figure 8.72 SATA cable

8.7.6 Interfacing to a PC

All of the PC I/O standards described so far are optimized for high performance and ease of attachment but are difficult to implement in hardware. Engineers and scientists often need a way to connect a PC to external circuitry, such as sensors, actuators, microcontrollers, or FPGAs. The serial connection described in Section 8.6.3.2 is sufficient for a low-speed connection to a microcontroller with a UART. This section describes two more means: data acquisition systems, and USB links.

Data Acquisition Systems

Data Acquisition Systems (DAQs) connect a computer to the real world using multiple channels of analog and/or digital I/O. DAQs are now

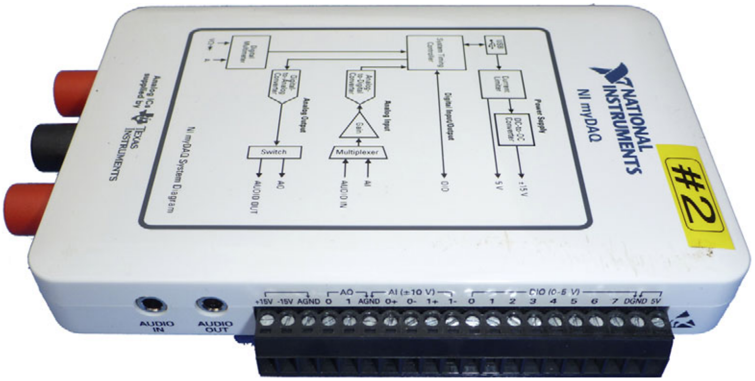


Figure 8.73 **NI myDAQ**

commonly available as USB devices, making them easy to install. National Instruments (NI) is a leading DAQ manufacturer.

High-performance DAQ prices tend to run into the thousands of dollars, mostly because the market is small and has limited competition. Fortunately, as of 2012, NI now sells their handy myDAQ system at a student discount price of \$200 including their LabVIEW software. Figure 8.73 shows a myDAQ. It has two analog channels capable of input and output at 200 ksamples/sec with a 16 bit resolution and ±10V dynamic range. These channels can be configured to operate as an oscilloscope and signal generator. It also has eight digital input and output lines compatible with 3.3 and 5 V systems. Moreover, it generates +5, +15, and –15 V power supply outputs and includes a digital multimeter capable of measuring voltage, current, and resistance. Thus, the myDAQ can replace an entire bench of test and measurement equipment while simultaneously offering automated data logging.

Most NI DAQs are controlled with LabVIEW, NI's graphical language for designing measurement and control systems. Some DAQs can also be controlled from C programs using the LabWindows environment, from Microsoft .NET applications using the Measurement Studio environment, or from Matlab using the Data Acquisition Toolbox.

USB Links

An increasing variety of products now provide simple, inexpensive digital links between PCs and external hardware over USB. These products contain predeveloped drivers and libraries, allowing the user to easily write a program on the PC that blasts data to and from an FPGA or microcontroller.

FTDI is a leading vendor for such systems. For example, the FTDI C232HM-DDHSL USB to Multi-Protocol Synchronous Serial Engine (MPSSE) cable shown in Figure 8.74 provides a USB jack at one end

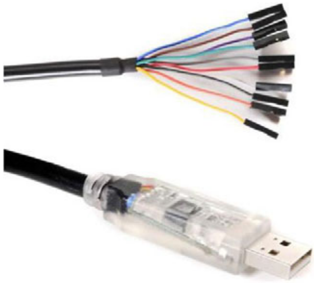


Figure 8.74 **FTDI USB to MPSSE cable**
(© 2012 by FTDI; reprinted with permission.)

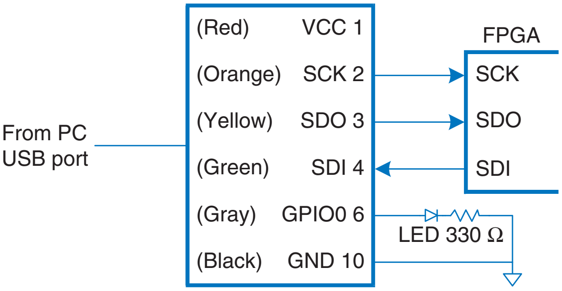


Figure 8.75 **C232HM-DDHSL USB to MPSESE interface from PC to FPGA**

and, at the other end, an SPI interface operating at up to 30 Mb/s, along with 3.3 V power and four general purpose I/O pins. Figure 8.75 shows an example of connecting a PC to an FPGA using the cable. The cable can optionally supply 3.3 V power to the FPGA. The three SPI pins connect to an FPGA slave device like the one from Example 8.19. The figure also shows one of the GPIO pins used to drive an LED.

The PC requires the D2XX dynamically linked library driver to be installed. You can then write a C program using the library to send data over the cable.

If an even faster connection is required, the FTDI UM232H module shown in Figure 8.76 links a PC's USB port to an 8-bit synchronous parallel interface operating up to 40 MB/s.

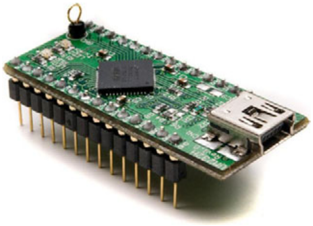


Figure 8.76 **FTDI UM232H module** (© 2012 by FTDI; reprinted with permission.)

8.8 REAL-WORLD PERSPECTIVE: x86 MEMORY AND I/O SYSTEMS*

As processors get faster, they need ever more elaborate memory hierarchies to keep a steady supply of data and instructions flowing. This section describes the memory systems of x86 processors to illustrate the progression. Section 7.9 contained photographs of the processors, highlighting the on-chip caches. x86 also has an unusual programmed I/O system that differs from the more common memory-mapped I/O.

8.8.1 x86 Cache Systems

The 80386, initially produced in 1985, operated at 16 MHz. It lacked a cache, so it directly accessed main memory for all instructions and data. Depending on the speed of the memory, the processor might get an immediate response, or it might have to pause for one or more cycles for the memory to react. These cycles are called *wait states*, and they increase the CPI of the processor. Microprocessor clock frequencies have increased by at least 25% per year since then, whereas memory latency

Table 8.13 **Evolution of Intel x86 microprocessor memory systems**

Processor	Year	Frequency (MHz)	Level 1 Data Cache	Level 1 Instruction Cache	Level 2 Cache
80386	1985	16–25	none	none	none
80486	1989	25–100	8 KB unified		none on chip
Pentium	1993	60–300	8 KB	8 KB	none on chip
Pentium Pro	1995	150–200	8 KB	8 KB	256 KB–1 MB on MCM
Pentium II	1997	233–450	16 KB	16 KB	256–512 KB on cartridge
Pentium III	1999	450–1400	16 KB	16 KB	256–512 KB on chip
Pentium 4	2001	1400–3730	8–16 KB	12 Kop trace cache	256 KB–2 MB on chip
Pentium M	2003	900–2130	32 KB	32 KB	1–2 MB on chip
Core Duo	2005	1500–2160	32 KB/core	32 KB/core	2 MB shared on chip
Core i7	2009	1600–3600	32 KB/core	32 KB/core	256 KB/core + 4-15 MB L3

has scarcely diminished. The delay from when the processor sends an address to main memory until the memory returns the data can now exceed 100 processor clock cycles. Therefore, caches with a low miss rate are essential to good performance. Table 8.13 summarizes the evolution of cache systems on Intel x86 processors.

The 80486 introduced a unified write-through cache to hold both instructions and data. Most high-performance computer systems also provided a larger second-level cache on the motherboard using commercially available SRAM chips that were substantially faster than main memory.

The Pentium processor introduced separate instruction and data caches to avoid contention during simultaneous requests for data and instructions. The caches used a write-back policy, reducing the communication with main memory. Again, a larger second-level cache (typically 256–512 KB) was usually offered on the motherboard.

The P6 series of processors (Pentium Pro, Pentium II, and Pentium III) were designed for much higher clock frequencies. The second-level cache on the motherboard could not keep up, so it was moved closer

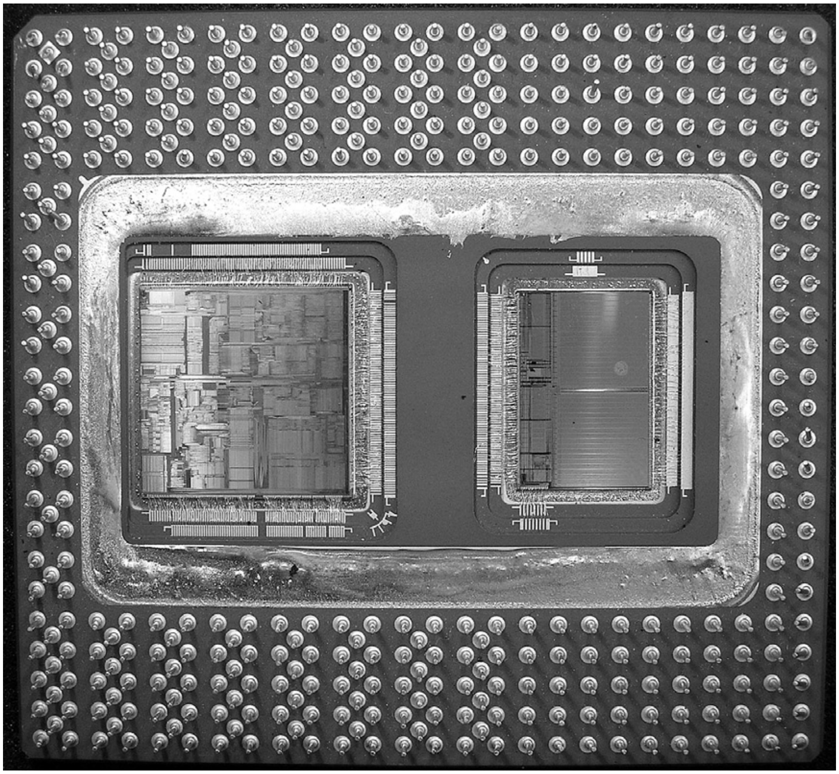


Figure 8.77 **Pentium Pro multichip module with processor (left) and 256-KB cache (right) in a pin grid array (PGA) package** (Courtesy Intel.)

to the processor to improve its latency and throughput. The Pentium Pro was packaged in a *multichip module* (*MCM*) containing both the processor chip and a second-level cache chip, as shown in Figure 8.77. Like the Pentium, the processor had separate 8-KB level 1 instruction and data caches. However, these caches were *nonblocking*, so that the out-of-order processor could continue executing subsequent cache accesses even if the cache missed a particular access and had to fetch data from main memory. The second-level cache was 256 KB, 512 KB, or 1 MB in size and could operate at the same speed as the processor. Unfortunately, the MCM packaging proved too expensive for high-volume manufacturing. Therefore, the Pentium II was sold in a lower-cost cartridge containing the processor and the second-level cache. The level 1 caches were doubled in size to compensate for the fact that the second-level cache operated at half the processor's speed. The Pentium III integrated a full-speed second-level cache directly onto the same chip as the processor. A cache on the same chip can operate at better latency and throughput, so it is substantially more effective than an off-chip cache of the same size.

The Pentium 4 offered a nonblocking level 1 data cache. It switched to a *trace cache* to store instructions after they had been decoded into micro-ops, avoiding the delay of redecoding each time instructions were fetched from the cache.

The Pentium M design was adapted from the Pentium III. It further increased the level 1 caches to 32 KB each and featured a 1- to 2-MB level 2 cache. The Core Duo contains two modified Pentium M processors and a shared 2-MB cache on one chip. The shared cache is used for communication between the processors: one can write data to the cache, and the other can read it.

The Nehalem (Core i3-i7) design adds a third level of cache shared between all of the cores on the die to facilitate sharing information between cores. Each core has its own 64 KB L1 cache and 256 KB L2 cache, while the shared L3 cache contains 4-8$^+$ MB.

Although memory protection became available in the hardware in the early 1980s, Microsoft Windows took almost 15 years to take advantage of the feature and prevent bad programs from crashing the entire computer. Until the release of Windows 2000, consumer versions of Windows were notoriously unstable. The lag between hardware features and software support can be extremely long.

8.8.2 x86 Virtual Memory

x86 processors operate in either real mode or protected mode. *Real mode* is backward compatible with the original 8086. It only uses 20 bits of address, limiting memory to 1 MB, and it does not allow virtual memory.

Protected mode was introduced with the 80286 and extended to 32-bit addresses with the 80386. It supports virtual memory with 4-KB pages. It also provides memory protection so that one program cannot access the pages belonging to other programs. Hence, a buggy or malicious program cannot crash or corrupt other programs. All modern operating systems now use protected mode.

A 32-bit address permits up to 4 GB of memory. Processors since the Pentium Pro have bumped the memory capacity to 64 GB using a technique called *physical address extension*. Each process uses 32-bit addresses. The virtual memory system maps these addresses onto a larger 36-bit virtual memory space. It uses different page tables for each process, so that each process can have its own address space of up to 4 GB.

To get around the memory bottleneck more gracefully, x86 has been upgraded to x86-64, which offers 64-bit virtual addresses and general-purpose registers. Presently, only 48 bits of the virtual address are used, providing a 256 terabyte (TB) virtual address space. The limit may be extended to the full 64 bits as memories expand, offering a 16 exabyte (EB) capacity.

8.8.3 x86 Programmed I/O

Most architectures use memory-mapped I/O, described in Section 8.5, in which programs access I/O devices by reading and writing memory locations. x86 uses *programmed* I/O, in which special IN and OUT instructions are used to read and write I/O devices. x86 defines 2^{16} I/O ports. The IN instruction reads one, two, or four bytes from the port specified by DX into AL, AX, or EAX. OUT is similar, but writes the port.

Connecting a peripheral device to a programmed I/O system is similar to connecting it to a memory-mapped system. When accessing an I/O port, the processor sends the port number rather than the memory address on the 16 least significant bits of the address bus. The device reads or writes data from the data bus. The major difference is that the processor also produces an $M/\overline{IO}$ signal. When $M/\overline{IO} = 1$, the processor is accessing memory. When it is 0, the process is accessing one of the I/O devices. The address decoder must also look at $M/\overline{IO}$ to generate the appropriate enables for main memory and for the I/O devices. I/O devices can also send interrupts to the processor to indicate that they are ready to communicate.

8.9 SUMMARY

Memory system organization is a major factor in determining computer performance. Different memory technologies, such as DRAM, SRAM, and hard drives, offer trade-offs in capacity, speed, and cost. This chapter introduced cache and virtual memory organizations that use a hierarchy of memories to approximate an ideal large, fast, inexpensive memory. Main memory is typically built from DRAM, which is significantly slower than the processor. A cache reduces access time by keeping commonly used data in fast SRAM. Virtual memory increases the memory capacity by using a hard drive to store data that does not fit in the main memory. Caches and virtual memory add complexity and hardware to a computer system, but the benefits usually outweigh the costs. All modern personal computers use caches and virtual memory. Most processors also use the memory interface to communicate with I/O devices. This is called memory-mapped I/O. Programs use load and store operations to access the I/O devices.

EPILOGUE

This chapter brings us to the end of our journey together into the realm of digital systems. We hope this book has conveyed the beauty and thrill of the art as well as the engineering knowledge. You have learned to design combinational and sequential logic using schematics and hardware description languages. You are familiar with larger building blocks such as multiplexers, ALUs, and memories. Computers are one of the most fascinating applications of digital systems. You have learned how to program a MIPS processor in its native assembly language and how to build the processor and memory system using digital building blocks. Throughout, you have seen the application of abstraction, discipline, hierarchy, modularity, and regularity. With these techniques, we have pieced together the puzzle of a microprocessor's inner workings. From cell phones to digital television to Mars rovers to medical imaging systems, our world is an increasingly digital place.

Imagine what Faustian bargain Charles Babbage would have made to take a similar journey a century and a half ago. He merely aspired to calculate mathematical tables with mechanical precision. Today's digital systems are yesterday's science fiction. Might Dick Tracy have listened to iTunes on his cell phone? Would Jules Verne have launched a constellation of global positioning satellites into space? Could Hippocrates have cured illness using high-resolution digital images of the brain? But at the same time, George Orwell's nightmare of ubiquitous government surveillance becomes closer to reality each day. Hackers and governments wage undeclared cyberwarfare, attacking industrial infrastructure and financial networks. And rogue states develop nuclear weapons using laptop computers more powerful than the room-sized supercomputers that simulated Cold War bombs. The microprocessor revolution continues to accelerate. The changes in the coming decades will surpass those of the past. You now have the tools to design and build these new systems that will shape our future. With your newfound power comes profound responsibility. We hope that you will use it, not just for fun and riches, but also for the benefit of humanity.

Exercises

Exercise 8.1 In less than one page, describe four everyday activities that exhibit temporal or spatial locality. List two activities for each type of locality, and be specific.

Exercise 8.2 In one paragraph, describe two short computer applications that exhibit temporal and/or spatial locality. Describe how. Be specific.

Exercise 8.3 Come up with a sequence of addresses for which a direct mapped cache with a size (capacity) of 16 words and block size of 4 words outperforms a fully associative cache with least recently used (LRU) replacement that has the same capacity and block size.

Exercise 8.4 Repeat Exercise 8.3 for the case when the fully associative cache outperforms the direct mapped cache.

Exercise 8.5 Describe the trade-offs of increasing each of the following cache parameters while keeping the others the same:

(a) block size

(b) associativity

(c) cache size

Exercise 8.6 Is the miss rate of a two-way set associative cache always, usually, occasionally, or never better than that of a direct mapped cache of the same capacity and block size? Explain.

Exercise 8.7 Each of the following statements pertains to the miss rate of caches. Mark each statement as true or false. Briefly explain your reasoning; present a counterexample if the statement is false.

(a) A two-way set associative cache always has a lower miss rate than a direct mapped cache with the same block size and total capacity.

(b) A 16-KB direct mapped cache always has a lower miss rate than an 8-KB direct mapped cache with the same block size.

(c) An instruction cache with a 32-byte block size usually has a lower miss rate than an instruction cache with an 8-byte block size, given the same degree of associativity and total capacity.

Exercise 8.8 A cache has the following parameters: b, block size given in numbers of words; S, number of sets; N, number of ways; and A, number of address bits.

(a) In terms of the parameters described, what is the cache capacity, C?

(b) In terms of the parameters described, what is the total number of bits required to store the tags?

(c) What are S and N for a fully associative cache of capacity C words with block size b?

(d) What is S for a direct mapped cache of size C words and block size b?

Exercise 8.9 A 16-word cache has the parameters given in Exercise 8.8. Consider the following repeating sequence of `lw` addresses (given in hexadecimal):

40 44 48 4C 70 74 78 7C 80 84 88 8C 90 94 98 9C 0 4 8 C 10 14 18 1C 20

Assuming least recently used (LRU) replacement for associative caches, determine the effective miss rate if the sequence is input to the following caches, ignoring startup effects (i.e., compulsory misses).

(a) direct mapped cache, $b = 1$ word

(b) fully associative cache, $b = 1$ word

(c) two-way set associative cache, $b = 1$ word

(d) direct mapped cache, $b = 2$ words

Exercise 8.10 Repeat Exercise 8.9 for the following repeating sequence of `lw` addresses (given in hexadecimal) and cache configurations. The cache capacity is still 16 words.

74 A0 78 38C AC 84 88 8C 7C 34 38 13C 388 18C

(a) direct mapped cache, $b = 1$ word

(b) fully associative cache, $b = 2$ words

(c) two-way set associative cache, $b = 2$ words

(d) direct mapped cache, $b = 4$ words

Exercise 8.11 Suppose you are running a program with the following data access pattern. The pattern is executed only once.

0x0 0x8 0x10 0x18 0x20 0x28

(a) If you use a direct mapped cache with a cache size of 1 KB and a block size of 8 bytes (2 words), how many sets are in the cache?

(b) With the same cache and block size as in part (a), what is the miss rate of the direct mapped cache for the given memory access pattern?

(c) For the given memory access pattern, which of the following would decrease the miss rate the most? (Cache capacity is kept constant.) Circle one.

(i) Increasing the degree of associativity to 2.

(ii) Increasing the block size to 16 bytes.

(iii) Either (i) or (ii).

(iv) Neither (i) nor (ii).

Exercise 8.12 You are building an instruction cache for a MIPS processor. It has a total capacity of $4C = 2^{c+2}$ bytes. It is $N = 2^n$-way set associative ($N \geq 8$), with a block size of $b = 2^{b'}$ bytes ($b \geq 8$). Give your answers to the following questions in terms of these parameters.

(a) Which bits of the address are used to select a word within a block?

(b) Which bits of the address are used to select the set within the cache?

(c) How many bits are in each tag?

(d) How many tag bits are in the entire cache?

Exercise 8.13 Consider a cache with the following parameters:
N (associativity) = 2, b (block size) = 2 words, W (word size) = 32 bits,
C (cache size) = 32 K words, A (address size) = 32 bits. You need consider only word addresses.

(a) Show the tag, set, block offset, and byte offset bits of the address. State how many bits are needed for each field.

(b) What is the size of *all* the cache tags in bits?

(c) Suppose each cache block also has a valid bit (V) and a dirty bit (D). What is the size of each cache set, including data, tag, and status bits?

(d) Design the cache using the building blocks in Figure 8.78 and a small number of two-input logic gates. The cache design must include tag storage, data

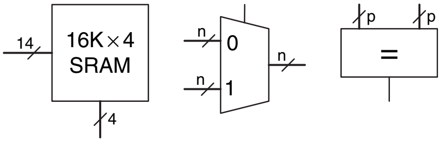


Figure 8.78 Building blocks

storage, address comparison, data output selection, and any other parts you feel are relevant. Note that the multiplexer and comparator blocks may be any size (n or p bits wide, respectively), but the SRAM blocks must be 16K × 4 bits. Be sure to include a neatly labeled block diagram. You need only design the cache for reads.

Exercise 8.14 You've joined a hot new Internet startup to build wrist watches with a built-in pager and Web browser. It uses an embedded processor with a multilevel cache scheme depicted in Figure 8.79. The processor includes a small on-chip cache in addition to a large off-chip second-level cache. (Yes, the watch weighs 3 pounds, but you should see it surf!)

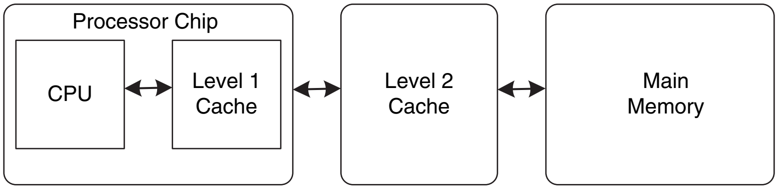


Figure 8.79 Computer system

Assume that the processor uses 32-bit physical addresses but accesses data only on word boundaries. The caches have the characteristics given in Table 8.14. The DRAM has an access time of t_m and a size of 512 MB.

Table 8.14 Memory characteristics

Characteristic	On-chip Cache	Off-chip Cache
Organization	Four-way set associative	Direct mapped
Hit rate	A	B
Access time	t_a	t_b
Block size	16 bytes	16 bytes
Number of blocks	512	256K

(a) For a given word in memory, what is the total number of locations in which it might be found in the on-chip cache and in the second-level cache?

(b) What is the size, in bits, of each tag for the on-chip cache and the second-level cache?

(c) Give an expression for the average memory read access time. The caches are accessed in sequence.

(d) Measurements show that, for a particular problem of interest, the on-chip cache hit rate is 85% and the second-level cache hit rate is 90%. However, when the on-chip cache is disabled, the second-level cache hit rate shoots up to 98.5%. Give a brief explanation of this behavior.

Exercise 8.15 This chapter described the least recently used (LRU) replacement policy for multiway associative caches. Other, less common, replacement policies include first-in-first-out (FIFO) and random policies. FIFO replacement evicts the block that has been there the longest, regardless of how recently it was accessed. Random replacement randomly picks a block to evict.

(a) Discuss the advantages and disadvantages of each of these replacement policies.

(b) Describe a data access pattern for which FIFO would perform better than LRU.

Exercise 8.16 You are building a computer with a hierarchical memory system that consists of separate instruction and data caches followed by main memory. You are using the MIPS multicycle processor from Figure 7.41 running at 1 GHz.

(a) Suppose the instruction cache is perfect (i.e., always hits) but the data cache has a 5% miss rate. On a cache miss, the processor stalls for 60 ns to access main memory, then resumes normal operation. Taking cache misses into account, what is the average memory access time?

(b) How many clock cycles per instruction (CPI) on average are required for load and store word instructions considering the non-ideal memory system?

(c) Consider the benchmark application of Example 7.7 that has 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions.[6] Taking the non-ideal memory system into account, what is the average CPI for this benchmark?

(d) Now suppose that the instruction cache is also non-ideal and has a 7% miss rate. What is the average CPI for the benchmark in part (c)? Take into account both instruction and data cache misses.

---

[6] Data from Patterson and Hennessy, *Computer Organization and Design*, 4th Edition, Morgan Kaufmann, 2011. Used with permission.

Exercise 8.17 Repeat Exercise 8.16 with the following parameters.

(a) The instruction cache is perfect (i.e., always hits) but the data cache has a 15% miss rate. On a cache miss, the processor stalls for 200 ns to access main memory, then resumes normal operation. Taking cache misses into account, what is the average memory access time?

(b) How many clock cycles per instruction (CPI) on average are required for load and store word instructions considering the non-ideal memory system?

(c) Consider the benchmark application of Example 7.7 that has 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions. Taking the non-ideal memory system into account, what is the average CPI for this benchmark?

(d) Now suppose that the instruction cache is also non-ideal and has a 10% miss rate. What is the average CPI for the benchmark in part (c)? Take into account both instruction and data cache misses.

Exercise 8.18 If a computer uses 64-bit virtual addresses, how much virtual memory can it access? Note that 2^{40} bytes = 1 *terabyte*, 2^{50} bytes = 1 *petabyte*, and 2^{60} bytes = 1 *exabyte*.

Exercise 8.19 A supercomputer designer chooses to spend \$1 million on DRAM and the same amount on hard disks for virtual memory. Using the prices from Figure 8.4, how much physical and virtual memory will the computer have? How many bits of physical and virtual addresses are necessary to access this memory?

Exercise 8.20 Consider a virtual memory system that can address a total of 2^{32} bytes. You have unlimited hard drive space, but are limited to only 8 MB of semiconductor (physical) memory. Assume that virtual and physical pages are each 4 KB in size.

(a) How many bits is the physical address?

(b) What is the maximum number of virtual pages in the system?

(c) How many physical pages are in the system?

(d) How many bits are the virtual and physical page numbers?

(e) Suppose that you come up with a direct mapped scheme that maps virtual pages to physical pages. The mapping uses the least significant bits of the virtual page number to determine the physical page number. How many virtual pages are mapped to each physical page? Why is this "direct mapping" a bad plan?

(f) Clearly, a more flexible and dynamic scheme for translating virtual addresses into physical addresses is required than the one described in part (e). Suppose you use a page table to store mappings (translations from virtual page number to physical page number). How many page table entries will the page table contain?

(g) Assume that, in addition to the physical page number, each page table entry also contains some status information in the form of a valid bit (*V*) and a dirty bit (*D*). How many bytes long is each page table entry? (Round up to an integer number of bytes.)

(h) Sketch the layout of the page table. What is the total size of the page table in bytes?

Exercise 8.21 Consider a virtual memory system that can address a total of 2^{50} bytes. You have unlimited hard drive space, but are limited to 2 GB of semiconductor (physical) memory. Assume that virtual and physical pages are each 4 KB in size.

(a) How many bits is the physical address?

(b) What is the maximum number of virtual pages in the system?

(c) How many physical pages are in the system?

(d) How many bits are the virtual and physical page numbers?

(e) How many page table entries will the page table contain?

(f) Assume that, in addition to the physical page number, each page table entry also contains some status information in the form of a valid bit (*V*) and a dirty bit (*D*). How many bytes long is each page table entry? (Round up to an integer number of bytes.)

(g) Sketch the layout of the page table. What is the total size of the page table in bytes?

Exercise 8.22 You decide to speed up the virtual memory system of Exercise 8.20 by using a translation lookaside buffer (TLB). Suppose your memory system has the characteristics shown in Table 8.15. The TLB and cache miss rates indicate

Table 8.15 Memory characteristics

Memory Unit	Access Time (Cycles)	Miss Rate
TLB	1	0.05%
Cache	1	2%
Main memory	100	0.0003%
Hard drive	1,000,000	0%

how often the requested entry is not found. The main memory miss rate indicates how often page faults occur.

(a) What is the average memory access time of the virtual memory system before and after adding the TLB? Assume that the page table is always resident in physical memory and is never held in the data cache.

(b) If the TLB has 64 entries, how big (in bits) is the TLB? Give numbers for data (physical page number), tag (virtual page number), and valid bits of each entry. Show your work clearly.

(c) Sketch the TLB. Clearly label all fields and dimensions.

(d) What size SRAM would you need to build the TLB described in part (c)? Give your answer in terms of depth × width.

Exercise 8.23 You decide to speed up the virtual memory system of Exercise 8.21 by using a translation lookaside buffer (TLB) with 128 entries.

(a) How big (in bits) is the TLB? Give numbers for data (physical page number), tag (virtual page number), and valid bits of each entry. Show your work clearly.

(b) Sketch the TLB. Clearly label all fields and dimensions.

(c) What size SRAM would you need to build the TLB described in part (b)? Give your answer in terms of depth × width.

Exercise 8.24 Suppose the MIPS multicycle processor described in Section 7.4 uses a virtual memory system.

(a) Sketch the location of the TLB in the multicycle processor schematic.

(b) Describe how adding a TLB affects processor performance.

Exercise 8.25 The virtual memory system you are designing uses a single-level page table built from dedicated hardware (SRAM and associated logic). It supports 25-bit virtual addresses, 22-bit physical addresses, and 2^{16}-byte (64 KB) pages. Each page table entry contains a physical page number, a valid bit (V), and a dirty bit (D).

(a) What is the total size of the page table, in bits?

(b) The operating system team proposes reducing the page size from 64 to 16 KB, but the hardware engineers on your team object on the grounds of added hardware cost. Explain their objection.

(c) The page table is to be integrated on the processor chip, along with the on-chip cache. The on-chip cache deals only with physical (not virtual) addresses. Is it possible to access the appropriate set of the on-chip cache concurrently with the page table access for a given memory access? Explain briefly the relationship that is necessary for concurrent access to the cache set and page table entry.

(d) Is it possible to perform the tag comparison in the on-chip cache concurrently with the page table access for a given memory access? Explain briefly.

Exercise 8.26 Describe a scenario in which the virtual memory system might affect how an application is written. Be sure to include a discussion of how the page size and physical memory size affect the performance of the application.

Exercise 8.27 Suppose you own a personal computer (PC) that uses 32-bit virtual addresses.

(a) What is the maximum amount of virtual memory space each program can use?

(b) How does the size of your PC's hard drive affect performance?

(c) How does the size of your PC's physical memory affect performance?

Exercise 8.28 Use MIPS memory-mapped I/O to interact with a user. Each time the user presses a button, a pattern of your choice displays on five light-emitting diodes (LEDs). Suppose the input button is mapped to address 0xFFFFFF10 and the LEDs are mapped to address 0xFFFFFF14. When the button is pushed, its output is 1; otherwise it is 0.

(a) Write MIPS code to implement this functionality.

(b) Draw a schematic for this memory-mapped I/O system.

(c) Write HDL code to implement the address decoder for your memory-mapped I/O system.

Exercise 8.29 Finite state machines (FSMs), like the ones you built in Chapter 3, can also be implemented in software.

(a) Implement the traffic light FSM from Figure 3.25 using MIPS assembly code. The inputs (T_A and T_B) are memory-mapped to bit 1 and bit 0, respectively, of address 0xFFFFF000. The two 3-bit outputs (L_A and L_B) are mapped to bits 0–2 and bits 3–5, respectively, of address 0xFFFFF004. Assume one-hot output encodings for each light, L_A and L_B; red is 100, yellow is 010, and green is 001.

(b) Draw a schematic for this memory-mapped I/O system.

(c) Write HDL code to implement the address decoder for your memory-mapped I/O system.

Exercise 8.30 Repeat Exercise 8.29 for the FSM in Figure 3.30(a). The input A and output Y are memory-mapped to bits 0 and 1, respectively, of address 0xFFFFF040.

Interview Questions

The following exercises present questions that have been asked on interviews.

Question 8.1 Explain the difference between direct mapped, set associative, and fully associative caches. For each cache type, describe an application for which that cache type will perform better than the other two.

Question 8.2 Explain how virtual memory systems work.

Question 8.3 Explain the advantages and disadvantages of using a virtual memory system.

Question 8.4 Explain how cache performance might be affected by the virtual page size of a memory system.

Question 8.5 Can addresses used for memory-mapped I/O be cached? Explain why or why not.

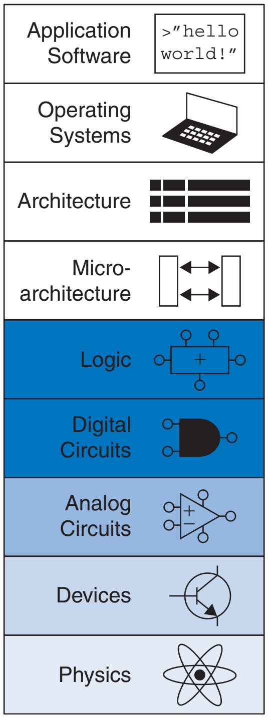
This page intentionally left blank

This page intentionally left blank

Digital System Implementation

A

A.1 Introduction
A.2 74xx Logic
A.3 Programmable Logic
A.4 Application-Specific Integrated Circuits
A.5 Data Sheets
A.6 Logic Families
A.7 Packaging and Assembly
A.8 Transmission Lines
A.9 Economics



A.1 INTRODUCTION

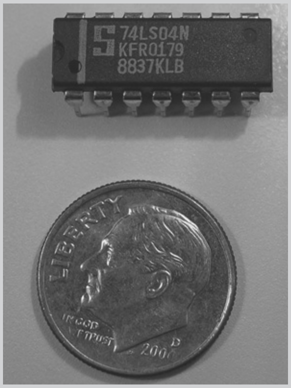
This appendix introduces practical issues in the design of digital systems. The material is not necessary for understanding the rest of the book, however, it seeks to demystify the process of building real digital systems. Moreover, we believe that the best way to understand digital systems is to build and debug them yourself in the laboratory.

Digital systems are usually built using one or more chips. One strategy is to connect together chips containing individual logic gates or larger elements such as arithmetic/logical units (ALUs) or memories. Another is to use programmable logic, which contains generic arrays of circuitry that can be programmed to perform specific logic functions. Yet a third is to design a custom integrated circuit containing the specific logic necessary for the system. These three strategies offer trade-offs in cost, speed, power consumption, and design time that are explored in the following sections. This appendix also examines the physical packaging and assembly of circuits, the transmission lines that connect the chips, and the economics of digital systems.

A.2 74xx LOGIC

In the 1970s and 1980s, many digital systems were built from simple chips, each containing a handful of logic gates. For example, the 7404 chip contains six NOT gates, the 7408 contains four AND gates, and the 7474 contains two flip-flops. These chips are collectively referred to as *74xx-series* logic. They were sold by many manufacturers, typically for 10 to 25 cents per chip. These chips are now largely obsolete, but they are still handy for simple digital systems or class projects, because they are so inexpensive and easy to use. 74xx-series chips are commonly sold in 14-pin *dual inline packages* (*DIPs*).

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00009-4
© 2013 Elsevier, Inc. All rights reserved.



74LS04 inverter chip in a 14-pin dual inline package. The part number is on the first line. LS indicates the logic family (see Section A.6). The N suffix indicates a DIP package. The large S is the logo of the manufacturer, Signetics. The bottom two lines of gibberish are codes indicating the batch in which the chip was manufactured.

A.2.1 Logic Gates

Figure A.1 shows the pinout diagrams for a variety of popular 74xx-series chips containing basic logic gates. These are sometimes called *small-scale integration* (*SSI*) chips, because they are built from a few transistors. The 14-pin packages typically have a notch at the top or a dot on the top left to indicate orientation. Pins are numbered starting with 1 in the upper left and going counterclockwise around the package. The chips need to receive power ($V_{DD} = 5$ V) and ground (GND = 0 V) at pins 14 and 7, respectively. The number of logic gates on the chip is determined by the number of pins. Note that pins 3 and 11 of the 7421 chip are not connected (NC) to anything. The 7474 flip-flop has the usual D, CLK, and Q terminals. It also has a complementary output, $\overline{Q}$. Moreover, it receives asynchronous set (also called preset, or PRE) and reset (also called clear, or CLR) signals. These are active low; in other words, the flop sets when $\overline{PRE} = 0$, resets when $\overline{CLR} = 0$, and operates normally when $\overline{PRE} = \overline{CLR} = 1$.

A.2.2 Other Functions

The 74xx series also includes somewhat more complex logic functions, including those shown in Figures A.2 and A.3. These are called *medium-scale integration* (*MSI*) chips. Most use larger packages to accommodate more inputs and outputs. Power and ground are still provided at the upper right and lower left, respectively, of each chip. A general functional description is provided for each chip. See the manufacturer's data sheets for complete descriptions.

A.3 PROGRAMMABLE LOGIC

Programmable logic consists of arrays of circuitry that can be configured to perform specific logic functions. We have already introduced three forms of programmable logic: programmable read only memories (PROMs), programmable logic arrays (PLAs), and field programmable gate arrays (FPGAs). This section shows chip implementations for each of these. Configuration of these chips may be performed by blowing on-chip fuses to connect or disconnect circuit elements. This is called *one-time programmable* (*OTP*) logic because, once a fuse is blown, it cannot be restored. Alternatively, the configuration may be stored in a memory that can be reprogrammed at will. Reprogrammable logic is convenient in the laboratory because the same chip can be reused during development.

A.3.1 PROMs

As discussed in Section 5.5.7, PROMs can be used as lookup tables. A 2^N-word × M-bit PROM can be programmed to perform any combinational function of N inputs and M outputs. Design changes simply involve

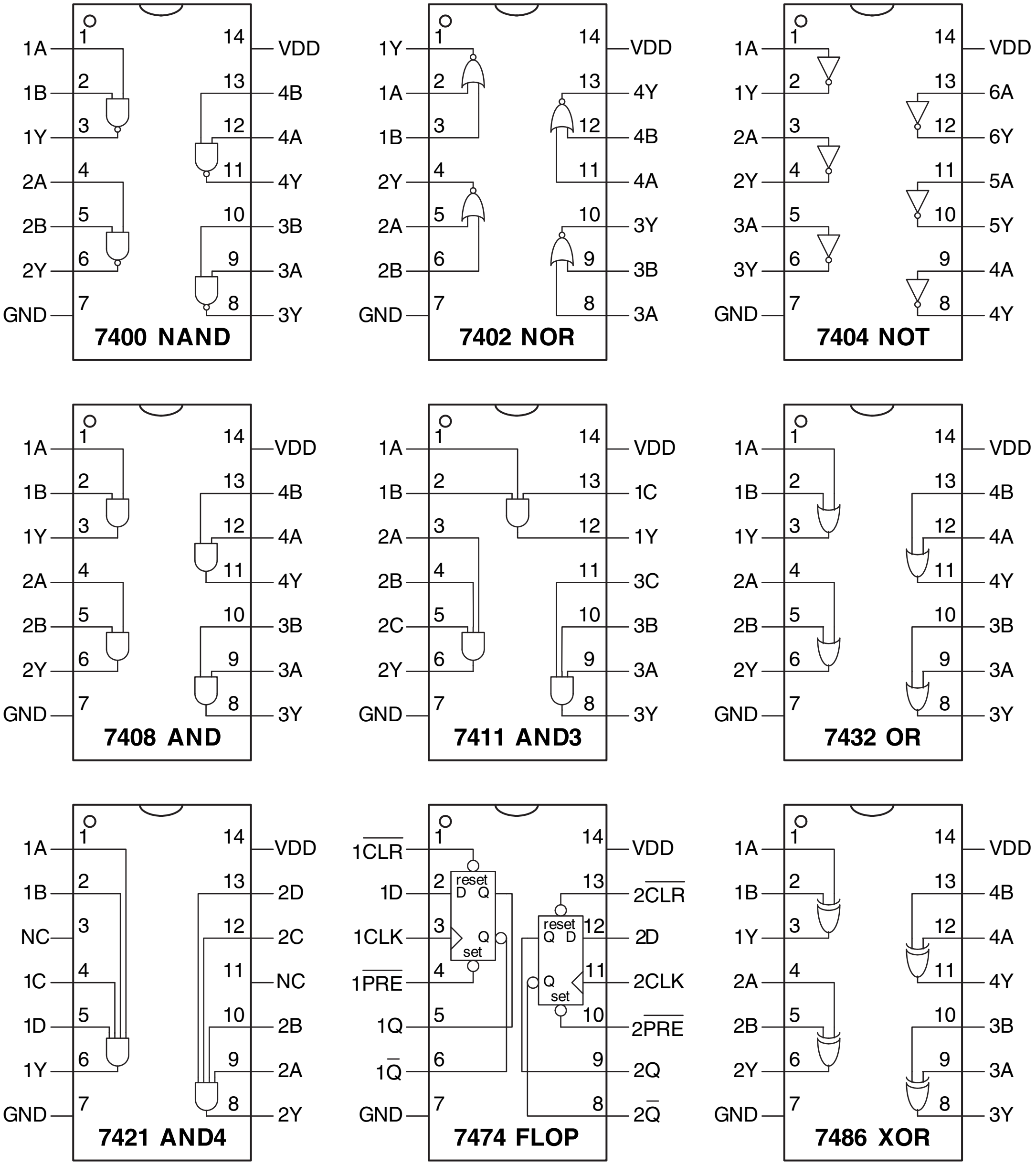
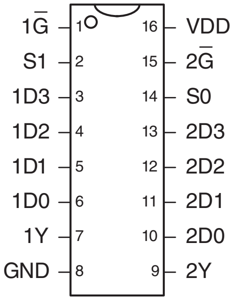

1A
1
14
VDD
1B
2
13
4B
1Y
3
12
4A
2A
4
11
4Y
2B
5
10
3B
2Y
6
9
3A
GND
7
8
3Y
7400 NAND
1Y
1
14
VDD
1A
2
13
4Y
1B
3
12
4B
2Y
4
11
4A
2A
5
10
3Y
2B
6
9
3B
GND
7
8
3A
7402 NOR
1A
1
14
VDD
1Y
2
13
6A
2A
3
12
6Y
2Y
4
11
5A
3A
5
10
5Y
3Y
6
9
4A
GND
7
8
4Y
7404 NOT
1A
1
14
VDD
1B
2
13
4B
1Y
3
12
4A
2A
4
11
4Y
2B
5
10
3B
2Y
6
9
3A
GND
7
8
3Y
7408 AND
1A
1
14
VDD
1B
2
13
1C
2A
3
12
1Y
2B
4
11
3C
2C
5
10
3B
2Y
6
9
3A
GND
7
8
3Y
7411 AND3
1A
1
14
VDD
1B
2
13
4B
1Y
3
12
4A
2A
4
11
4Y
2B
5
10
3B
2Y
6
9
3A
GND
7
8
3Y
7432 OR
1A
1
14
VDD
1B
2
13
2D
NC
3
12
2C
1C
4
11
NC
1D
5
10
2B
1Y
6
9
2A
GND
7
8
2Y
7421 AND4
$1\overline{\text{CLR}}$
1
14
VDD
1D
2
reset
D Q
13
$2\overline{\text{CLR}}$
reset
Q D
1CLK
3
Q
set
12
2D
$1\overline{\text{PRE}}$
4
Q
set
11
2CLK
1Q
5
10
$2\overline{\text{PRE}}$
$1\overline{\text{Q}}$
6
9
2Q
GND
7
8
$2\overline{\text{Q}}$
7474 FLOP
1A
1
14
VDD
1B
2
13
4B
1Y
3
12
4A
2A
4
11
4Y
2B
5
10
3B
2Y
6
9
3A
GND
7
8
3Y
7486 XOR

Figure A.1 Common 74xx-series logic gates



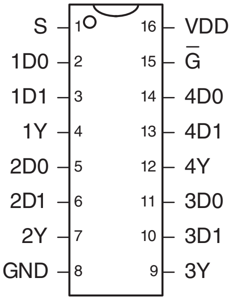
$\overline{1G}$ 1 16 VDD
S1 2 15 $\overline{2G}$
1D3 3 14 S0
1D2 4 13 2D3
1D1 5 12 2D2
1D0 6 11 2D1
1Y 7 10 2D0
GND 8 9 2Y

74153 4:1 Mux

Two 4:1 Multiplexers

$D_{3:0}$:	data
$S_{1:0}$:	select
Y:	output
Gb:	enable

```
always_comb
  if (1Gb) 1Y = 0;
  else     1Y = 1D[S];
always_comb
  if (2Gb) 2Y = 0;
  else     2Y = 2D[S];
```



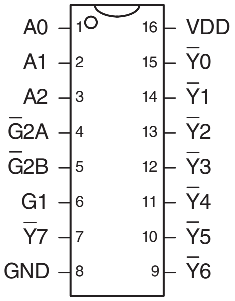
S 1 16 VDD
1D0 2 15 $\overline{G}$
1D1 3 14 4D0
1Y 4 13 4D1
2D0 5 12 4Y
2D1 6 11 3D0
2Y 7 10 3D1
GND 8 9 3Y

74157 2:1 Mux

Four 2:1 Multiplexers

$D_{1:0}$:	data
S:	select
Y:	output
Gb:	enable

```
always_comb
  if  (Gb) 1Y = 0;
  else     1Y = S ? 1D[1]  : 1D[0];
  if  (Gb) 2Y = 0;
  else     2Y = S ? 2D[1]  : 2D[0];
  if  (Gb) 3Y = 0;
  else     3Y = S ? 3D[1]  : 3D[0];
  if  (Gb) 4Y = 0;
  else     4Y = S ? 4D[1]  : 4D[0];
```



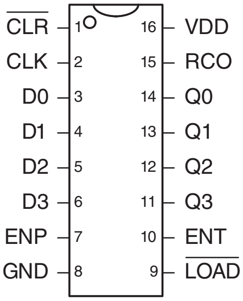
A0 1 16 VDD
A1 2 15 $\overline{Y0}$
A2 3 14 $\overline{Y1}$
$\overline{G2A}$ 4 13 $\overline{Y2}$
$\overline{G2B}$ 5 12 $\overline{Y3}$
G1 6 11 $\overline{Y4}$
$\overline{Y7}$ 7 10 $\overline{Y5}$
GND 8 9 $\overline{Y6}$

74138 3:8 Decoder

3:8 Decoder

$A_{2:0}$:	address
$Yb_{7:0}$:	output
G1:	active high enable
G2:	active low enables

G1	$\overline{G2A}$	$\overline{G2B}$	A2:0	Y7:0
0	x	x	xxx	11111111
1	1	x	xxx	11111111
1	0	1	xxx	11111111
1	0	0	000	11111110
1	0	0	001	11111101
1	0	0	010	11111011
1	0	0	011	11110111
1	0	0	100	11101111
1	0	0	101	11011111
1	0	0	110	10111111
1	0	0	111	01111111



$\overline{CLR}$ 1 16 VDD
CLK 2 15 RCO
D0 3 14 Q0
D1 4 13 Q1
D2 5 12 Q2
D3 6 11 Q3
ENP 7 10 ENT
GND 8 9 $\overline{LOAD}$

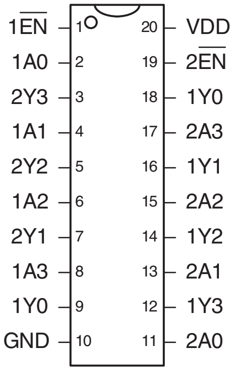
74161/163 Counter

4-bit Counter

CLK:	clock
$Q_{3:0}$:	counter output
$D_{3:0}$:	parallel input
CLRb:	async reset (161) sync reset (163)
LOADb:	load Q from D
ENP, ENT:	enables
RCO:	ripple carry out

```
always @(posedge CLK) // 74163
  if (~CLRb) Q <= 4'b0000;
  else if (~LOADb) Q <= D;
  else if (ENP & ENT) Q <= Q+1;

assign RCO = (Q == 4'b1111) & ENT;
```



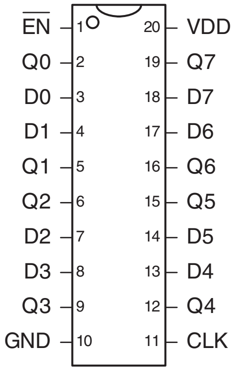
$\overline{1EN}$ 1 20 VDD
1A0 2 19 $\overline{2EN}$
2Y3 3 18 1Y0
1A1 4 17 2A3
2Y2 5 16 1Y1
1A2 6 15 2A2
2Y1 7 14 1Y2
1A3 8 13 2A1
1Y0 9 12 1Y3
GND 10 11 2A0

74244 Tristate Buffer

8-bit Tristate Buffer

$A_{3:0}$:	input
$Y_{3:0}$:	output
ENb:	enable

```
assign 1Y =
  1ENb ? 4'bzzzz : 1A;
assign 2Y =
  2ENb ? 4'bzzzz : 2A;
```



$\overline{EN}$ 1 20 VDD
Q0 2 19 Q7
D0 3 18 D7
D1 4 17 D6
Q1 5 16 Q6
Q2 6 15 Q5
D2 7 14 D5
D3 8 13 D4
Q3 9 12 Q4
GND 10 11 CLK

74377 Register

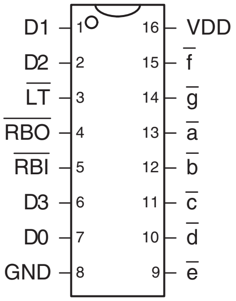
8-bit Enableable Register

CLK:	clock
$D_{7:0}$:	data
$Q_{7:0}$:	output
ENb:	enable

```
always_ff @(posedge CLK)
  if (~ENb) Q <= D;
```

Note: SystemVerilog variable names cannot start with numbers, but the names in the example code in Figure A.2 are chosen to match the manufacturer's data sheet.

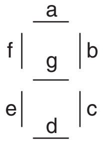
Figure A.2 Medium-scale integration chips


D1
D2
$\overline{LT}$
$\overline{RBO}$
$\overline{RBI}$
D3
D0
GND
VDD
$\overline{f}$
$\overline{g}$
$\overline{a}$
$\overline{b}$
$\overline{c}$
$\overline{d}$
$\overline{e}$
1
2
3
4
5
6
7
8
16
15
14
13
12
11
10
9

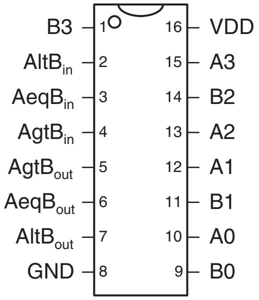
7447 7-Segment Decoder

7-segment Display Decoder

$D_{3:0}$:	data
a...f :	segments (low = ON)
LTb:	light test
RBIb:	ripple blanking in
RBOb:	ripple blanking out



$\overline{RBO}$	$\overline{LT}$	$\overline{RBI}$	D3:0	$\overline{a}$	$\overline{b}$	$\overline{c}$	$\overline{d}$	$\overline{e}$	$\overline{f}$	$\overline{g}$
0	x	x	x	1	1	1	1	1	1	1
1	0	x	x	0	0	0	0	0	0	0
x	1	0	0000	1	1	1	1	1	1	1
1	1	1	0000	0	0	0	0	0	0	1
1	1	1	0001	1	0	0	1	1	1	1
1	1	1	0010	0	0	1	0	0	1	0
1	1	1	0011	0	0	0	0	1	1	0
1	1	1	0100	1	0	0	1	1	0	0
1	1	1	0101	0	1	0	0	1	0	0
1	1	1	0110	1	1	0	0	0	0	0
1	1	1	0111	0	0	0	1	1	1	1
1	1	1	1000	0	0	0	0	0	0	0
1	1	1	1001	0	0	0	1	1	0	0
1	1	1	1010	1	1	1	0	0	1	0
1	1	1	1011	1	1	0	0	1	1	0
1	1	1	1100	1	0	1	1	1	0	0
1	1	1	1101	0	1	1	0	1	0	0
1	1	1	1110	0	0	0	1	1	1	1
1	1	1	1111	0	0	0	0	0	0	0

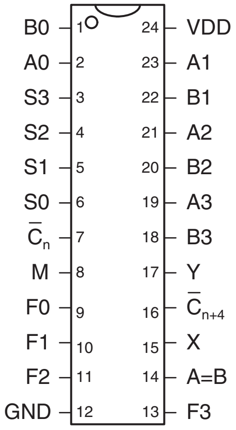

B3
$AltB_{in}$
$AeqB_{in}$
$AgtB_{in}$
$AgtB_{out}$
$AeqB_{out}$
$AltB_{out}$
GND
VDD
A3
B2
A2
A1
B1
A0
B0
1
2
3
4
5
6
7
8
16
15
14
13
12
11
10
9

7485 Comparator

4-bit Comparator

$A_{3:0}$, $B_{3:0}$:	data
rel_{in} :	input relation
rel_{out} :	output relation

```
always_comb
   if (A > B | (A == B & AgtBin)) begin
     AgtBout = 1; AeqBout = 0; AltBout = 0;
   end
   else if (A < B | (A == B & AltBin) begin
     AgtBout = 0; AeqBout = 0; AltBout = 1;
   end else begin
     AgtBout = 0; AeqBout = 1; AltBout = 0;
   end
```


B0
A0
S3
S2
S1
S0
$\overline{C}_n$
M
F0
F1
F2
GND
VDD
A1
B1
A2
B2
A3
B3
Y
$\overline{C}_{n+4}$
X
A=B
F3
1
2
3
4
5
6
7
8
9
10
11
12
24
23
22
21
20
19
18
17
16
15
14
13

74181 ALU

4-bit ALU

$A_{3:0}$, $B_{3:0}$:	inputs
$Y_{3:0}$:	output
$F_{3:0}$:	function select
M :	mode select
Cb_n :	carry in
Cb_{nplus4} :	carry out
AeqB :	equality (in some modes)
X,Y :	carry lookahead adder outputs

```
always_comb
  case (F)
    0000: Y = M ? ~A         :  A          + ~Cbn;
    0001: Y = M ? ~(A | B)   :  A       +  B          + ~Cbn;
    0010: Y = M ? (~A) & B   :  A       + ~B          + ~Cbn;
    0011: Y = M ? 4'b0000    :  4'b1111              + ~Cbn;
    0100: Y = M ? ~(A & B)   :  A       + (A & ~B)   + ~Cbn;
    0101: Y = M ? ~B         : (A | B)  + (A & ~B)   + ~Cbn;
    0110: Y = M ? A ^ B      :  A       - B          -  Cbn;
    0111: Y = M ? A & ~B     : (A & ~B)              -  Cbn;
    1000: Y = M ? ~A + B     :  A       + (A & B)    + ~Cbn;
    1001: Y = M ? ~(A ^ B)   :  A       +  B          + ~Cbn;
    1010: Y = M ? B          : (A | ~B) + (A & B)    + ~Cbn;
    1011: Y = M ? A & B      : (A & B)               + ~Cbn;
    1100: Y = M ? 1          :  A       +  A          + ~Cbn;
    1101: Y = M ? A | ~B     : (A | B)  +  A          + ~Cbn;
    1110: Y = M ? A | B      : (A | ~B) +  A          + ~Cbn;
    1111: Y = M ? A          :  A                     -  Cbn;
  endcase
```

Figure A.3 More medium-scale integration (MSI) chips

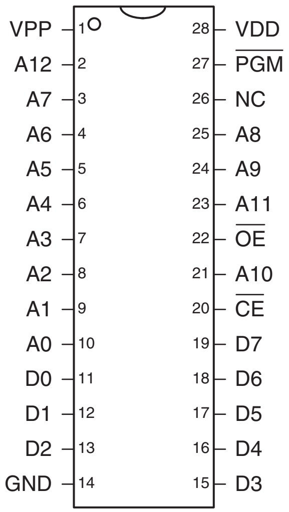


Figure A.4 2764 8KB EPROM

8 KB EPROM

$A_{12:0}$:	address input
$D_{7:0}$:	data output
CEb :	chip enable
OEb :	output enable
PGMb :	program
VPP :	program voltage
NC :	no connection

```
assign D = (~CEb & ~OEb) ? ROM[A]
                         : 8'bz;
```

replacing the contents of the PROM rather than rewiring connections between chips. Lookup tables are useful for small functions but become prohibitively expensive as the number of inputs grows.

For example, the classic 2764 8-KB (64-Kb) erasable PROM (EPROM) is shown in Figure A.4. The EPROM has 13 address lines to specify one of the 8K words and 8 data lines to read the byte of data at that word. The chip enable and output enable must both be asserted for data to be read. The maximum propagation delay is 200 ps. In normal operation, $\overline{PGM} = 1$ and VPP is not used. The EPROM is usually programmed on a special programmer that sets $\overline{PGM} = 0$, applies 13 V to VPP, and uses a special sequence of inputs to configure the memory.

Modern PROMs are similar in concept but have much larger capacities and more pins. Flash memory is the cheapest type of PROM, selling for about \$1 per gigabyte in 2012. Prices have historically declined by 30 to 40% per year.

A.3.2 PLAs

As discussed in Section 5.6.1, PLAs contain AND and OR planes to compute any combinational function written in sum-of-products form. The AND and OR planes can be programmed using the same techniques for PROMs. A PLA has two columns for each input and one column for each output. It has one row for each minterm. This organization is more efficient than a PROM for many functions, but the array still grows excessively large for functions with numerous I/Os and minterms.

Many different manufacturers have extended the basic PLA concept to build *programmable logic devices* (*PLDs*) that include registers. The 22V10 is one of the most popular classic PLDs. It has 12 dedicated input pins and 10 outputs. The outputs can come directly from the PLA or from clocked registers on the chip. The outputs can also be fed back into the PLA. Thus, the 22V10 can directly implement FSMs with up to 12 inputs, 10 outputs, and 10 bits of state. The 22V10 costs about $2 in quantities of 100. PLDs have been rendered mostly obsolete by the rapid improvements in capacity and cost of FPGAs.

A.3.3 FPGAs

As discussed in Section 5.6.2, FPGAs consist of arrays of configurable *logic elements* (*LEs*), also called *configurable logic blocks* (*CLBs*), connected together with programmable wires. The LEs contain small lookup tables and flip-flops. FPGAs scale gracefully to extremely large capacities, with thousands of lookup tables. Xilinx and Altera are two of the leading FPGA manufacturers.

Lookup tables and programmable wires are flexible enough to implement any logic function. However, they are an order of magnitude less efficient in speed and cost (chip area) than hard-wired versions of the same functions. Thus, FPGAs often include specialized blocks, such as memories, multipliers, and even entire microprocessors.

Figure A.5 shows the design process for a digital system on an FPGA. The design is usually specified with a hardware description language (HDL), although some FPGA tools also support schematics. The design is then simulated. Inputs are applied and compared against expected outputs to *verify* that the logic is correct. Usually some debugging is required. Next, logic *synthesis* converts the HDL into Boolean functions. Good synthesis tools produce a schematic of the functions, and the prudent designer examines these schematics, as well as any warnings produced during synthesis, to ensure that the desired logic was produced. Sometimes sloppy coding leads to circuits that are much larger than intended or to circuits with asynchronous logic. When the synthesis results are good, the FPGA tool *maps* the functions onto the LEs of a specific chip. The *place and route* tool determines which functions go in which lookup tables and how they are wired together. Wire delay increases with length, so critical circuits should be placed close together. If the design is too big to fit on the chip, it must be reengineered. *Timing analysis* compares the timing constraints (e.g., an intended clock speed of 100 MHz) against the actual circuit delays and reports any errors. If the logic is too slow, it may have to be redesigned or pipelined differently. When the design is correct, a file is generated specifying the contents of all the LEs and the programming of all the wires on the FPGA. Many FPGAs store this *configuration* information in static RAM that must be reloaded each time the FPGA is

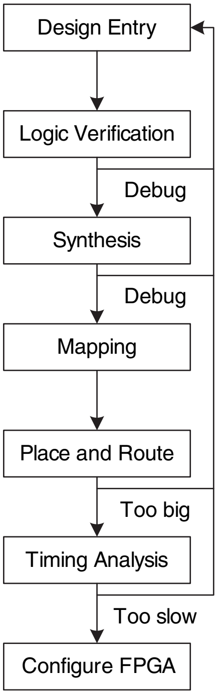


Figure A.5 **FPGA design flow**

turned on. The FPGA can download this information from a computer in the laboratory, or can read it from a nonvolatile ROM when power is first applied.

Example A.1 FPGA TIMING ANALYSIS

Alyssa P. Hacker is using an FPGA to implement an M&M sorter with a color sensor and motors to put red candy in one jar and green candy in another. Her design is implemented as an FSM, and she is using a Cyclone IV GX. According to the data sheet, the FPGA has the timing characteristics shown in Table A.1.

Alyssa would like her FSM to run at 100 MHz. What is the maximum number of LEs on the critical path? What is the fastest speed at which her FSM could possibly run?

Solution: At 100 MHz, the cycle time, T_c, is 10 ns. Alyssa uses Equation 3.13 to figure out the minimum combinational propagation delay, t_{pd}, at this cycle time:

$$t_{pd} \le 10\,\text{ns} - (0.199\,\text{ns} + 0.076\,\text{ns}) = 9.725\,\text{ns} \tag{A.1}$$

With a combined LE and wire delay of 381 ps + 246 ps = 627 ps, Alyssa's FSM can use at most 15 consecutive LEs (9.725/0.627) to implement the next-state logic.

The fastest speed at which an FSM will run on this Cyclone IV FPGA is when it is using a single LE for the next state logic. The minimum cycle time is

$$T_c \ge 381\,\text{ps} + 199\,\text{ps} + 76\,\text{ps} = 656\,\text{ps} \tag{A.2}$$

Therefore, the maximum frequency is 1.5 GHz.

Table A.1 Cyclone IV GX timing

name	value (ps)
t_{pcq}	199
t_{setup}	76
t_{hold}	0
t_{pd} (per LE)	381
t_{wire} (between LEs)	246
t_{skew}	0

Altera advertises the Cyclone IV FPGA with 14,400 LEs for \$25 in 2012. In large quantities, medium-sized FPGAs typically cost several dollars. The largest FPGAs cost hundreds or even thousands of dollars.

The cost has declined at approximately 30% per year, so FPGAs are becoming extremely popular.

A.4 APPLICATION-SPECIFIC INTEGRATED CIRCUITS

Application-specific integrated circuits (*ASICs*) are chips designed for a particular purpose. Graphics accelerators, network interface chips, and cell phone chips are common examples of ASICS. The ASIC designer places transistors to form logic gates and wires the gates together. Because the ASIC is hardwired for a specific function, it is typically several times faster than an FPGA and occupies an order of magnitude less chip area (and hence cost) than an FPGA with the same function. However, the *masks* specifying where transistors and wires are located on the chip cost hundreds of thousands of dollars to produce. The fabrication process usually requires 6 to 12 weeks to manufacture, package, and test the ASICs. If errors are discovered after the ASIC is manufactured, the designer must correct the problem, generate new masks, and wait for another batch of chips to be fabricated. Hence, ASICs are suitable only for products that will be produced in large quantities and whose function is well defined in advance.

Figure A.6 shows the ASIC design process, which is similar to the FPGA design process of Figure A.5. Logic verification is especially important because correction of errors after the masks are produced is expensive. Synthesis produces a *netlist* consisting of logic gates and connections between the gates; the gates in this netlist are placed, and the wires are routed between gates. When the design is satisfactory, masks are generated and used to fabricate the ASIC. A single speck of dust can ruin an ASIC, so the chips must be tested after fabrication. The fraction of manufactured chips that work is called the *yield*; it is typically 50 to 90%, depending on the size of the chip and the maturity of the manufacturing process. Finally, the working chips are placed in packages, as will be discussed in Section A.7.

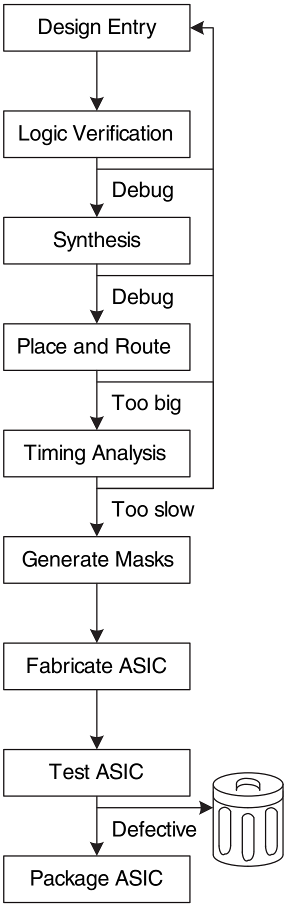


Figure A.6 ASIC design flow

A.5 DATA SHEETS

Integrated circuit manufacturers publish *data sheets* that describe the functions and performance of their chips. It is essential to read and understand the data sheets. One of the leading sources of errors in digital systems comes from misunderstanding the operation of a chip.

Data sheets are usually available from the manufacturer's Web site. If you cannot locate the data sheet for a part and do not have clear documentation from another source, don't use the part. Some of the entries in the data sheet may be cryptic. Often the manufacturer publishes data books containing data sheets for many related parts. The beginning of the data book has additional explanatory information. This information can usually be found on the Web with a careful search.

This section dissects the Texas Instruments (TI) data sheet for a 74HC04 inverter chip. The data sheet is relatively simple but illustrates many of the major elements. TI still manufacturers a wide variety of 74xx-series chips. In the past, many other companies built these chips too, but the market is consolidating as the sales decline.

Figure A.7 shows the first page of the data sheet. Some of the key sections are highlighted in blue. The title is SN54HC04, SN74HC04 HEX INVERTERS. HEX INVERTERS means that the chip contains six inverters. SN indicates that TI is the manufacturer. Other manufacture codes include MC for Motorola and DM for National Semiconductor. You can generally ignore these codes, because all of the manufacturers build compatible 74xx-series logic. HC is the logic family (high speed CMOS). The logic family determines the speed and power consumption of the chip, but not the function. For example, the 7404, 74HC04, and 74LS04 chips all contain six inverters, but they differ in performance and cost. Other logic families are discussed in Section A.6. The 74xx chips operate across the commercial or industrial temperature range (0 to 70 °C or −40 to 85 °C, respectively), whereas the 54xx chips operate across the military temperature range (−55 to 125 °C) and sell for a higher price but are otherwise compatible.

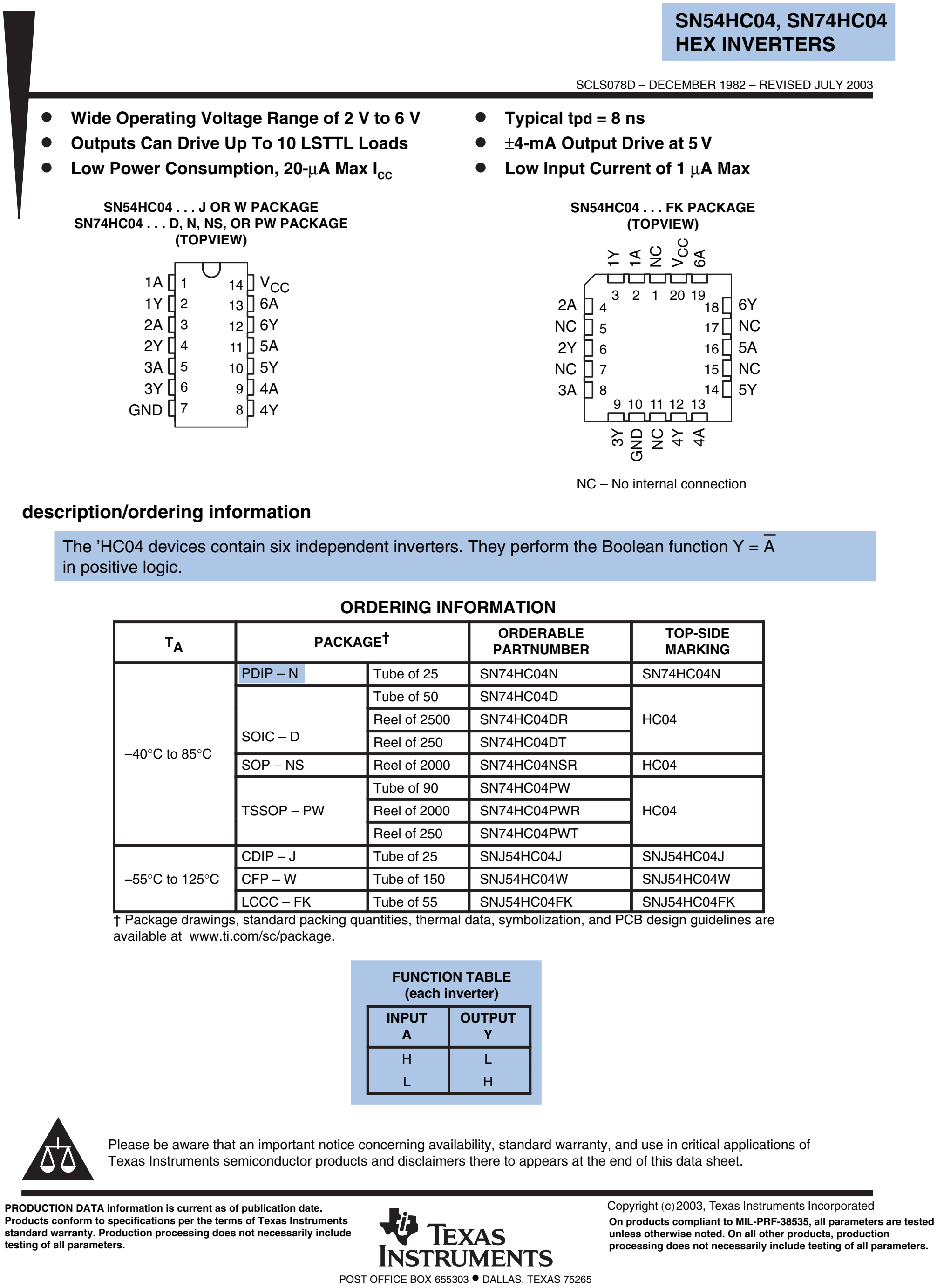
The 7404 is available in many different packages, and it is important to order the one you intended when you make a purchase. The packages are distinguished by a suffix on the part number. N indicates a *plastic dual inline package* (*PDIP*), which fits in a breadboard or can be soldered in through-holes in a printed circuit board. Other packages are discussed in Section A.7.

The function table shows that each gate inverts its input. If A is HIGH (H), Y is LOW (L) and vice versa. The table is trivial in this case but is more interesting for more complex chips.

Figure A.8 shows the second page of the data sheet. The logic diagram indicates that the chip contains inverters. The *absolute maximum* section indicates conditions beyond which the chip could be destroyed. In particular, the power supply voltage (V_{CC}, also called V_{DD} in this book) should not exceed 7 V. The continuous output current should not exceed 25 mA. The *thermal resistance* or impedance, θ_{JA}, is used to calculate the temperature rise caused by the chip's dissipating power. If the *ambient* temperature in the vicinity of the chip is T_A and the chip dissipates P_{chip}, then the temperature on the chip itself at its *junction* with the package is

$$T_J = T_A + P_{\text{chip}}\,\theta_{JA} \tag{A.3}$$

For example, if a 7404 chip in a plastic DIP package is operating in a hot box at 50 °C and consumes 20 mW, the junction temperature will climb



SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

- **Wide Operating Voltage Range of 2 V to 6 V**
- **Outputs Can Drive Up To 10 LSTTL Loads**
- **Low Power Consumption, 20-µA Max I_{CC}**
- **Typical tpd = 8 ns**
- **±4-mA Output Drive at 5 V**
- **Low Input Current of 1 µA Max**

SN54HC04 . . . J OR W PACKAGE
SN74HC04 . . . D, N, NS, OR PW PACKAGE
(TOPVIEW)

1A 1
1Y 2
2A 3
2Y 4
3A 5
3Y 6
GND 7
14 VCC
13 6A
12 6Y
11 5A
10 5Y
9 4A
8 4Y

SN54HC04 . . . FK PACKAGE
(TOPVIEW)

1Y 1A NC VCC 6A
3 2 1 20 19
2A 4
NC 5
2Y 6
NC 7
3A 8
18 6Y
17 NC
16 5A
15 NC
14 5Y
9 10 11 12 13
3Y GND NC 4Y 4A

NC – No internal connection

description/ordering information

The ’HC04 devices contain six independent inverters. They perform the Boolean function Y = $\overline{A}$ in positive logic.

ORDERING INFORMATION

T_A	PACKAGE†		ORDERABLE PARTNUMBER	TOP-SIDE MARKING
–40°C to 85°C	PDIP – N	Tube of 25	SN74HC04N	SN74HC04N
	SOIC – D	Tube of 50	SN74HC04D	HC04
		Reel of 2500	SN74HC04DR	
		Reel of 250	SN74HC04DT	
	SOP – NS	Reel of 2000	SN74HC04NSR	HC04
	TSSOP – PW	Tube of 90	SN74HC04PW	HC04
		Reel of 2000	SN74HC04PWR	
		Reel of 250	SN74HC04PWT	
–55°C to 125°C	CDIP – J	Tube of 25	SNJ54HC04J	SNJ54HC04J
	CFP – W	Tube of 150	SNJ54HC04W	SNJ54HC04W
	LCCC – FK	Tube of 55	SNJ54HC04FK	SNJ54HC04FK

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.

FUNCTION TABLE
(each inverter)

INPUT A	OUTPUT Y
H	L
L	H

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers there to appears at the end of this data sheet.

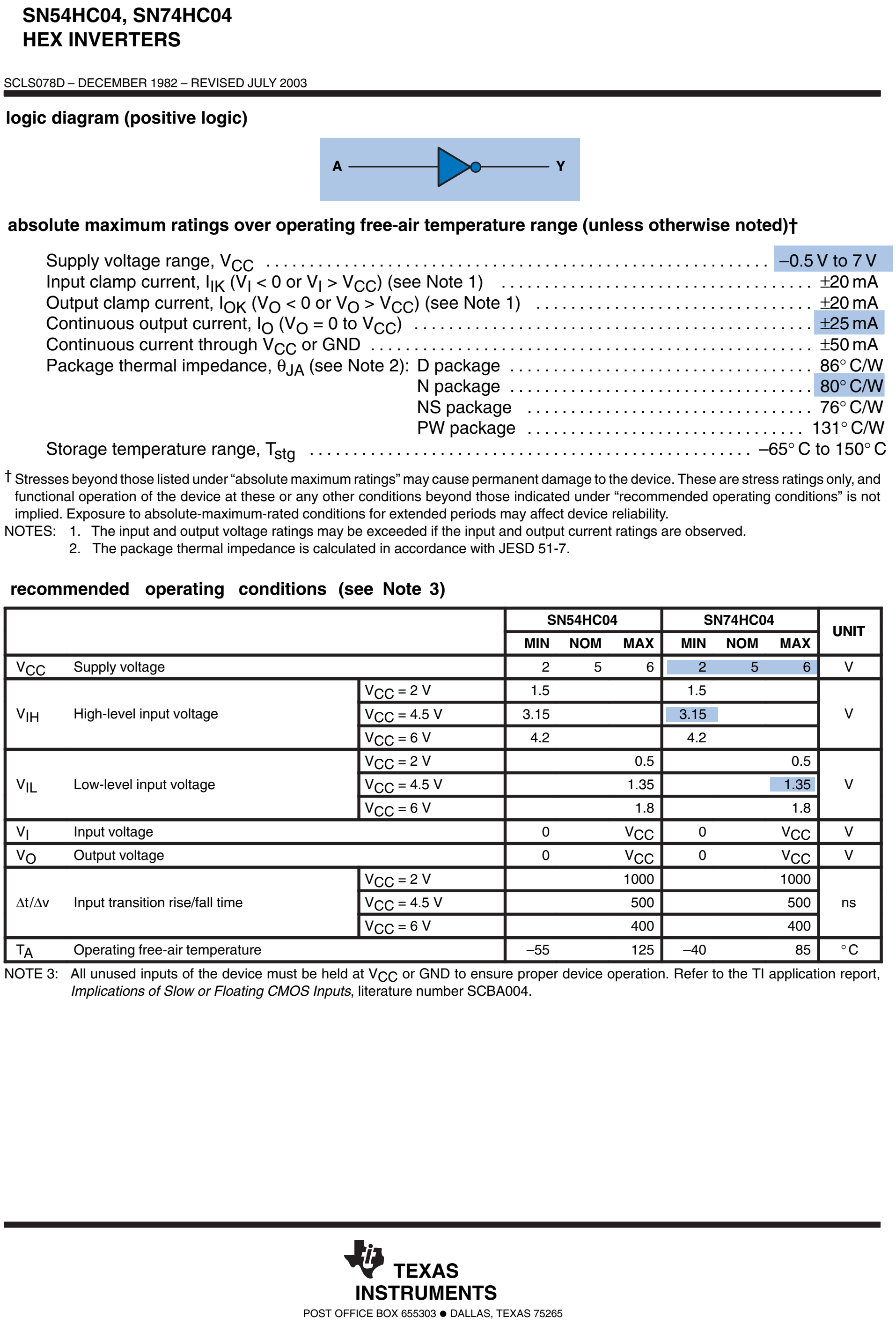
PRODUCTION DATA information is current as of publication date. Products conform to specifications per the terms of Texas Instruments standard warranty. Production processing does not necessarily include testing of all parameters.

Copyright (c) 2003, Texas Instruments Incorporated

On products compliant to MIL-PRF-38535, all parameters are tested unless otherwise noted. On all other products, production processing does not necessarily include testing of all parameters.

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

Figure A.7 **7404 data sheet page 1**



SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

logic diagram (positive logic)

absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Supply voltage range, V_{CC}	–0.5 V to 7 V
Input clamp current, I_{IK} ($V_I < 0$ or $V_I > V_{CC}$) (see Note 1)	±20 mA
Output clamp current, I_{OK} ($V_O < 0$ or $V_O > V_{CC}$) (see Note 1)	±20 mA
Continuous output current, I_O ($V_O = 0$ to V_{CC})	±25 mA
Continuous current through V_{CC} or GND	±50 mA
Package thermal impedance, θ_{JA} (see Note 2): D package	86° C/W
N package	80° C/W
NS package	76° C/W
PW package	131° C/W
Storage temperature range, T_{stg}	–65° C to 150° C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTES: 1. The input and output voltage ratings may be exceeded if the input and output current ratings are observed.
2. The package thermal impedance is calculated in accordance with JESD 51-7.

recommended operating conditions (see Note 3)

			SN54HC04			SN74HC04			UNIT
			MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage		2	5	6	2	5	6	V
V_{IH}	High-level input voltage	$V_{CC} = 2$ V	1.5			1.5			V
		$V_{CC} = 4.5$ V	3.15			3.15			
		$V_{CC} = 6$ V	4.2			4.2			
V_{IL}	Low-level input voltage	$V_{CC} = 2$ V			0.5			0.5	V
		$V_{CC} = 4.5$ V			1.35			1.35	
		$V_{CC} = 6$ V			1.8			1.8	
V_I	Input voltage		0		V_{CC}	0		V_{CC}	V
V_O	Output voltage		0		V_{CC}	0		V_{CC}	V
$\Delta t/\Delta v$	Input transition rise/fall time	$V_{CC} = 2$ V			1000			1000	ns
		$V_{CC} = 4.5$ V			500			500	
		$V_{CC} = 6$ V			400			400	
T_A	Operating free-air temperature		–55		125	–40		85	°C

NOTE 3: All unused inputs of the device must be held at V_{CC} or GND to ensure proper device operation. Refer to the TI application report, *Implications of Slow or Floating CMOS Inputs*, literature number SCBA004.

TEXAS
INSTRUMENTS
POST OFFICE BOX 655303 ● DALLAS, TEXAS 75265

Figure A.8 **7404 data sheet page 2**

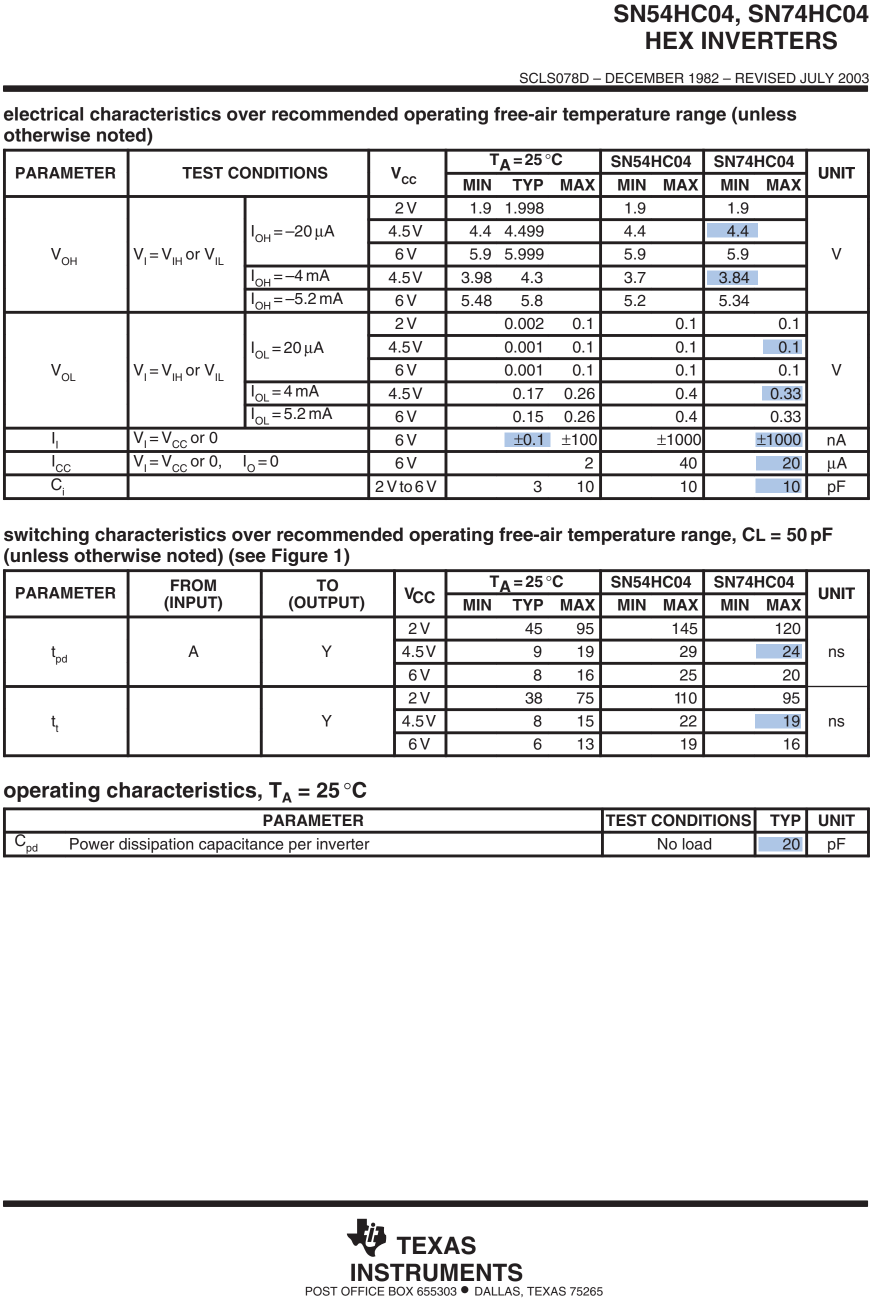
to 50 °C + 0.02 W × 80 °C/W = 51.6 °C. Internal power dissipation is seldom important for 74xx-series chips, but it becomes important for modern chips that dissipate tens of watts or more.

The *recommended operating conditions* define the environment in which the chip should be used. Within these conditions, the chip should meet specifications. These conditions are more stringent than the absolute maximums. For example, the power supply voltage should be between 2 and 6 V. The input logic levels for the HC logic family depend on V_{DD}. Use the 4.5 V entries when $V_{DD} = 5$ V, to allow for a 10% droop in the power supply caused by noise in the system.

Figure A.9 shows the third page of the data sheet. The *electrical characteristics* describe how the device performs when used within the recommended operating conditions if the inputs are held constant. For example, if $V_{CC} = 5$ V (and droops to 4.5 V) and the output current I_{OH}/I_{OL} does not exceed 20 μA, $V_{OH} = 4.4$ V and $V_{OL} = 0.1$ V in the worst case. If the output current increases, the output voltages become less ideal, because the transistors on the chip struggle to provide the current. The HC logic family uses CMOS transistors that draw very little current. The current into each input is guaranteed to be less than 1000 nA and is typically only 0.1 nA at room temperature. The *quiescent* power supply current (I_{DD}) drawn while the chip is idle is less than 20 μA. Each input has less than 10 pF of capacitance.

The *switching characteristics* define how the device performs when used within the recommended operating conditions if the inputs change. The *propagation delay*, t_{pd}, is measured from when the input passes through 0.5 V_{CC} to when the output passes through 0.5 V_{CC}. If V_{CC} is nominally 5 V and the chip drives a capacitance of less than 50 pF, the propagation delay will not exceed 24 ns (and typically will be much faster). Recall that each input may present 10 pF, so the chip cannot drive more than five identical chips at full speed. Indeed, stray capacitance from the wires connecting chips cuts further into the useful load. The *transition time*, also called the rise/fall time, is measured as the output transitions between 0.1 V_{CC} and 0.9 V_{CC}.

Recall from Section 1.8 that chips consume both *static* and *dynamic power*. Static power is low for HC circuits. At 85 °C, the maximum quiescent supply current is 20 μA. At 5 V, this gives a static power consumption of 0.1 mW. The dynamic power depends on the capacitance being driven and the switching frequency. The 7404 has an internal power dissipation capacitance of 20 pF per inverter. If all six inverters on the 7404 switch at 10 MHz and drive external loads of 25 pF, then the dynamic power given by Equation 1.4 is $\frac{1}{2}(6)(20\text{ pF} + 25\text{ pF})(5^2)(10\text{ MHz}) = 33.75$ mW and the maximum total power is 33.85 mW.



SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER	TEST CONDITIONS		V_{CC}	T_A = 25 °C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
V_{OH}	$V_I = V_{IH}$ or V_{IL}	$I_{OH} = -20\,\mu A$	2 V	1.9	1.998		1.9		1.9		V
			4.5 V	4.4	4.499		4.4		4.4		
			6 V	5.9	5.999		5.9		5.9		
		$I_{OH} = -4$ mA	4.5 V	3.98	4.3		3.7		3.84		
		$I_{OH} = -5.2$ mA	6 V	5.48	5.8		5.2		5.34		
V_{OL}	$V_I = V_{IH}$ or V_{IL}	$I_{OL} = 20\,\mu A$	2 V		0.002	0.1		0.1		0.1	V
			4.5 V		0.001	0.1		0.1		0.1	
			6 V		0.001	0.1		0.1		0.1	
		$I_{OL} = 4$ mA	4.5 V		0.17	0.26		0.4		0.33	
		$I_{OL} = 5.2$ mA	6 V		0.15	0.26		0.4		0.33	
I_I	$V_I = V_{CC}$ or 0		6 V		±0.1	±100		±1000		±1000	nA
I_{CC}	$V_I = V_{CC}$ or 0, $I_O = 0$		6 V			2		40		20	µA
C_i			2 V to 6 V		3	10		10		10	pF

switching characteristics over recommended operating free-air temperature range, CL = 50 pF (unless otherwise noted) (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	V_{CC}	T_A = 25 °C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
t_{pd}	A	Y	2 V		45	95		145		120	ns
			4.5 V		9	19		29		24	
			6 V		8	16		25		20	
t_t		Y	2 V		38	75		110		95	ns
			4.5 V		8	15		22		19	
			6 V		6	13		19		16	

operating characteristics, T_A = 25 °C

PARAMETER		TEST CONDITIONS	TYP	UNIT
C_{pd}	Power dissipation capacitance per inverter	No load	20	pF

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

Figure A.9 7404 data sheet page 3

A.6 LOGIC FAMILIES

The 74xx-series logic chips have been manufactured using many different technologies, called *logic families*, that offer different speed, power, and logic level trade-offs. Other chips are usually designed to be compatible with some of these logic families. The original chips, such as the 7404, were built using bipolar transistors in a technology called *Transistor-Transistor Logic* (*TTL*). Newer technologies add one or more letters after the 74 to indicate the logic family, such as 74LS04, 74HC04, or 74AHCT04. Table A.2 summarizes the most common 5-V logic families.

Advances in bipolar circuits and process technology led to the *Schottky* (*S*) and *Low-Power Schottky* (*LS*) families. Both are faster than TTL. Schottky draws more power, whereas Low-Power Schottky draws less. *Advanced Schottky* (*AS*) and *Advanced Low-Power Schottky* (*ALS*) have improved speed and power compared to S and LS. *Fast* (*F*) logic is faster and draws less power than AS. All of these families provide more current for LOW outputs than for HIGH outputs and hence have

Table A.2 **Typical specifications for 5-V logic families**

	Bipolar / TTL						CMOS		CMOS / TTL Compatible	
Characteristic	TTL	S	LS	AS	ALS	F	HC	AHC	HCT	AHCT
t_{pd} (ns)	22	9	12	7.5	10	6	21	7.5	30	7.7
V_{IH} (V)	2	2	2	2	2	2	3.15	3.15	2	2
V_{IL} (V)	0.8	0.8	0.8	0.8	0.8	0.8	1.35	1.35	0.8	0.8
V_{OH} (V)	2.4	2.7	2.7	2.5	2.5	2.5	3.84	3.8	3.84	3.8
V_{OL} (V)	0.4	0.5	0.5	0.5	0.5	0.5	0.33	0.44	0.33	0.44
I_{OH} (mA)	0.4	1	0.4	2	0.4	1	4	8	4	8
I_{OL} (mA)	16	20	8	20	8	20	4	8	4	8
I_{IL} (mA)	1.6	2	0.4	0.5	0.1	0.6	0.001	0.001	0.001	0.001
I_{IH} (mA)	0.04	0.05	0.02	0.02	0.02	0.02	0.001	0.001	0.001	0.001
I_{DD} (mA)	33	54	6.6	26	4.2	15	0.02	0.02	0.02	0.02
C_{Pd} (pF)	n/a						20	12	20	14
cost* (US $)	obsolete	0.63	0.25	0.53	0.32	0.22	0.12	0.12	0.12	0.12

* Per unit in quantities of 1000 for the 7408 from Texas Instruments in 2012

asymmetric logic levels. They conform to the "TTL" logic levels: $V_{IH} = 2$ V, $V_{IL} = 0.8$ V, $V_{OH} > 2.4$ V, and $V_{OL} < 0.5$ V.

As CMOS circuits matured in the 1980s and 1990s, they became popular because they draw very little power supply or input current. The *High Speed CMOS* (*HC*) and *Advanced High Speed CMOS* (*AHC*) families draw almost no static power. They also deliver the same current for HIGH and LOW outputs. They conform to the "CMOS" logic levels: $V_{IH} = 3.15$ V, $V_{IL} = 1.35$ V, $V_{OH} > 3.8$ V, and $V_{OL} < 0.44$ V. Unfortunately, these levels are incompatible with TTL circuits, because a TTL HIGH output of 2.4 V may not be recognized as a legal CMOS HIGH input. This motivates the use of *High Speed TTL-compatible CMOS* (*HCT*) and *Advanced High Speed TTL-compatible CMOS* (*AHCT*), which accept TTL input logic levels and generate valid CMOS output logic levels. These families are slightly slower than their pure CMOS counterparts. All CMOS chips are sensitive to *electrostatic discharge* (*ESD*) caused by static electricity. Ground yourself by touching a large metal object before handling CMOS chips, lest you zap them.

The 74xx-series logic is inexpensive. The newer logic families are often cheaper than the obsolete ones. The LS family is widely available and robust and is a popular choice for laboratory or hobby projects that have no special performance requirements.

The 5-V standard collapsed in the mid-1990s, when transistors became too small to withstand the voltage. Moreover, lower voltage offers lower power consumption. Now 3.3, 2.5, 1.8, 1.2, and even lower voltages are commonly used. The plethora of voltages raises challenges in communicating between chips with different power supplies. Table A.3 lists some of the low-voltage logic families. Not all 74xx parts are available in all of these logic families.

All of the low-voltage logic families use CMOS transistors, the workhorse of modern integrated circuits. They operate over a wide range of V_{DD}, but the speed degrades at lower voltage. *Low-Voltage CMOS* (*LVC*) logic and *Advanced Low-Voltage CMOS* (*ALVC*) logic are commonly used at 3.3, 2.5, or 1.8 V. LVC withstands inputs up to 5.5 V, so it can receive inputs from 5-V CMOS or TTL circuits. *Advanced Ultra-Low-Voltage CMOS* (*AUC*) is commonly used at 2.5, 1.8, or 1.2 V and is exceptionally fast. Both ALVC and AUC withstand inputs up to 3.6 V, so they can receive inputs from 3.3-V circuits.

FPGAs often offer separate voltage supplies for the internal logic, called the *core*, and for the input/output (I/O) pins. As FPGAs have advanced, the core voltage has dropped from 5 to 3.3, 2.5, 1.8, and 1.2 V to save power and avoid damaging the very small transistors. FPGAs have configurable I/Os that can operate at many different voltages, so as to be compatible with the rest of the system.

Table A.3 Typical specifications for low-voltage logic families

	LVC			ALVC			AUC		
V_{dd} (V)	3.3	2.5	1.8	3.3	2.5	1.8	2.5	1.8	1.2
t_{pd} (ns)	4.1	6.9	9.8	2.8	3	?[1]	1.8	2.3	3.4
V_{IH} (V)	2	1.7	1.17	2	1.7	1.17	1.7	1.17	0.78
V_{IL} (V)	0.8	0.7	0.63	0.8	0.7	0.63	0.7	0.63	0.42
V_{OH} (V)	2.2	1.7	1.2	2	1.7	1.2	1.8	1.2	0.8
V_{OL} (V)	0.55	0.7	0.45	0.55	0.7	0.45	0.6	0.45	0.3
I_O (mA)	24	8	4	24	12	12	9	8	3
I_I (mA)	0.02			0.005			0.005		
I_{DD} (mA)	0.01			0.01			0.01		
C_{pd} (pF)	10	9.8	7	27.5	23	?*	17	14	14
cost (US $)	0.17			0.20			not available		

* Delay and capacitance not available at the time of writing

A.7 PACKAGING AND ASSEMBLY

Integrated circuits are typically placed in *packages* made of plastic or ceramic. The packages serve a number of functions, including connecting the tiny metal I/O pads of the chip to larger pins in the package for ease of connection, protecting the chip from physical damage, and spreading the heat generated by the chip over a larger area to help with cooling. The packages are placed on a breadboard or printed circuit board and wired together to assemble the system.

Packages

Figure A.10 shows a variety of integrated circuit packages. Packages can be generally categorized as *through-hole* or *surface mount* (*SMT*). Through-hole packages, as their name implies, have pins that can be inserted through holes in a printed circuit board or into a socket. *Dual inline packages* (*DIPs*) have two rows of pins with 0.1-inch spacing between pins. *Pin grid arrays* (*PGAs*) support more pins in a smaller package by placing the pins under the package. SMT packages are soldered directly to the surface of a printed circuit board without using holes. Pins on SMT parts are called *leads*. The *thin small outline package* (*TSOP*) has two rows of closely spaced leads (typically 0.02-inch spacing). *Plastic leaded chip carriers* (*PLCCs*) have J-shaped leads on all four

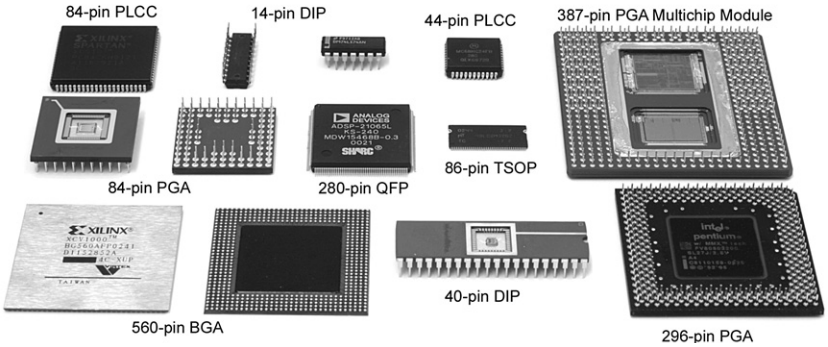


Figure A.10 **Integrated circuit packages**

sides, with 0.05-inch spacing. They can be soldered directly to a board or placed in special sockets. *Quad flat packs* (*QFPs*) accommodate a large number of pins using closely spaced legs on all four sides. *Ball grid arrays* (*BGAs*) eliminate the legs altogether. Instead, they have hundreds of tiny solder balls on the underside of the package. They are carefully placed over matching pads on a printed circuit board, then heated so that the solder melts and joins the package to the underlying board.

Breadboards

DIPs are easy to use for prototyping, because they can be placed in a *breadboard*. A breadboard is a plastic board containing rows of sockets, as shown in Figure A.11. All five holes in a row are connected together. Each pin of the package is placed in a hole in a separate row. Wires can be placed in adjacent holes in the same row to make connections to the pin. Breadboards often provide separate columns of connected holes running the height of the board to distribute power and ground.

Figure A.11 shows a breadboard containing a majority gate built with a 74LS08 AND chip and a 74LS32 OR chip. The schematic of the circuit is shown in Figure A.12. Each gate in the schematic is labeled with the chip (08 or 32) and the pin numbers of the inputs and outputs (see Figure A.1). Observe that the same connections are made on the breadboard. The inputs are connected to pins 1, 2, and 5 of the 08 chip, and the output is measured at pin 6 of the 32 chip. Power and ground are connected to pins 14 and 7, respectively, of each chip, from the vertical power and ground columns that are attached to the banana plug receptacles, Vb and Va. Labeling the schematic in this way and checking off connections as they are made is a good way to reduce the number of mistakes made during breadboarding.

Unfortunately, it is easy to accidentally plug a wire in the wrong hole or have a wire fall out, so breadboarding requires a great deal of care (and usually some debugging in the laboratory). Breadboards are suited only to prototyping, not production.

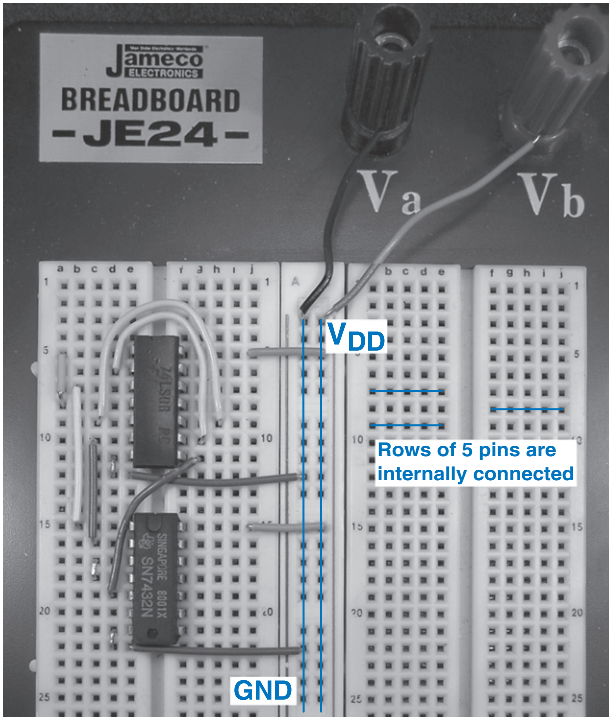

Jameco ELECTRONICS
BREADBOARD
-JE24-
Va
Vb
VDD
Rows of 5 pins are internally connected
GND
74LS08
SN7432N

Figure A.11 Majority circuit on breadboard

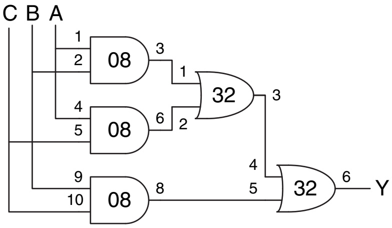

C B A
1
2
08
3
1
32
3
4
5
08
6
2
4
9
10
08
8
5
32
6
Y

Figure A.12 Majority gate schematic with chips and pins identified

Printed Circuit Boards

Instead of breadboarding, chip packages may be soldered to a *printed circuit board* (*PCB*). The PCB is formed of alternating layers of conducting copper and insulating epoxy. The copper is etched to form wires called *traces*. Holes called *vias* are drilled through the board and plated with metal to connect between layers. PCBs are usually designed with *computer-aided design* (*CAD*) tools. You can etch and drill your own simple boards in the laboratory, or you can send the board design to a specialized factory for inexpensive mass production. Factories have turnaround times of days (or weeks, for cheap mass production runs) and typically charge a few hundred dollars in setup fees and a few dollars per board for moderately complex boards built in large quantities.

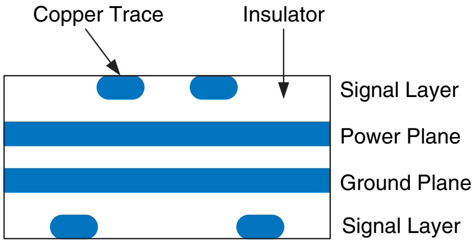


Figure A.13 **Printed circuit board cross-section**

PCB traces are normally made of copper because of its low resistance. The traces are embedded in an insulating material, usually a green, fire-resistant plastic called FR4. A PCB also typically has copper power and ground layers, called *planes*, between signal layers. Figure A.13 shows a cross-section of a PCB. The signal layers are on the top and bottom, and the power and ground planes are embedded in the center of the board. The power and ground planes have low resistance, so they distribute stable power to components on the board. They also make the capacitance and inductance of the traces uniform and predictable.

Figure A.14 shows a PCB for a 1970s vintage Apple II+ computer. At the top is a 6502 microprocessor. Beneath are six 16-Kb ROM chips forming 12 KB of ROM containing the operating system. Three rows of eight 16-Kb DRAM chips provide 48 KB of RAM. On the right are several rows of 74xx-series logic for memory address decoding and other functions. The lines between chips are traces that wire the chips together. The dots at the ends of some of the traces are vias filled with metal.

Putting It All Together

Most modern chips with large numbers of inputs and outputs use SMT packages, especially QFPs and BGAs. These packages require a printed circuit board rather than a breadboard. Working with BGAs is especially challenging because they require specialized assembly equipment. Moreover, the balls cannot be probed with a voltmeter or oscilloscope during debugging in the laboratory, because they are hidden under the package.

In summary, the designer needs to consider packaging early on to determine whether a breadboard can be used during prototyping and whether BGA parts will be required. Professional engineers rarely use breadboards when they are confident of connecting chips together correctly without experimentation.

A.8 TRANSMISSION LINES

We have assumed so far that wires are *equipotential* connections that have a single voltage along their entire length. Signals actually propagate along wires at the speed of light in the form of electromagnetic waves. If the wires are short enough or the signals change slowly, the equipotential

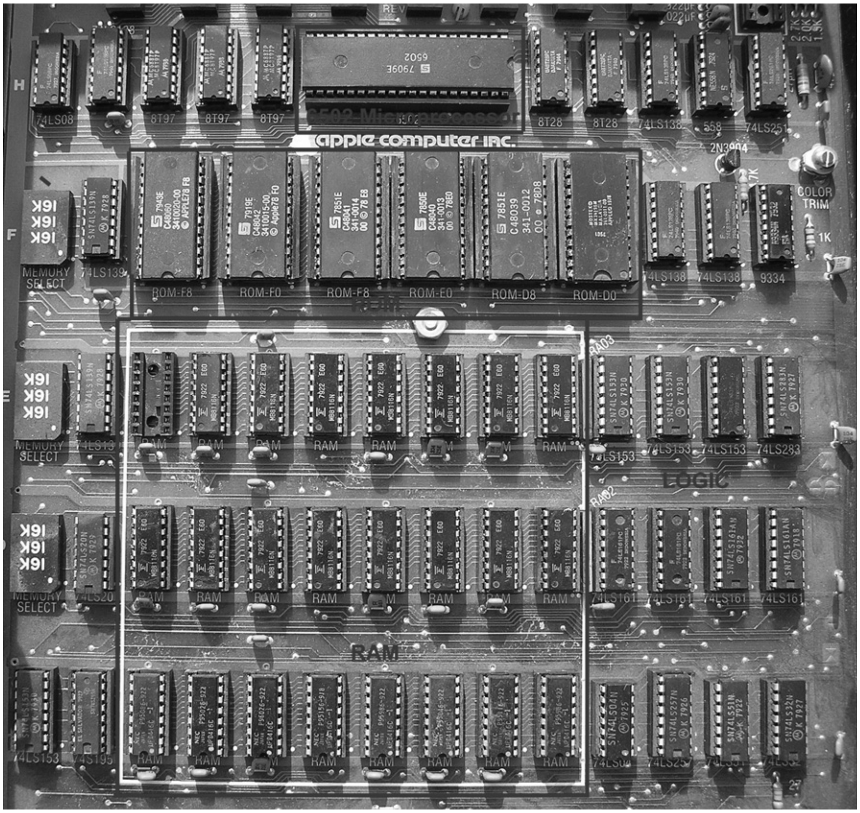


Figure A.14 **Apple II+ circuit board**

assumption is good enough. When the wire is long or the signal is very fast, the *transmission time* along the wire becomes important to accurately determine the circuit delay. We must model such wires as *transmission lines*, in which a wave of voltage and current propagates at the speed of light. When the wave reaches the end of the line, it may reflect back along the line. The reflection may cause noise and odd behaviors unless steps are taken to limit it. Hence, the digital designer must consider transmission line behavior to accurately account for the delay and noise effects in long wires.

Electromagnetic waves travel at the speed of light in a given medium, which is fast but not instantaneous. The speed of light, ν, depends on the *permittivity*, ε, and *permeability*, μ, of the medium[1]: $\nu = \frac{1}{\sqrt{\mu\varepsilon}} = \frac{1}{\sqrt{LC}}$.

---

[1] The capacitance, C, and inductance, L, of a wire are related to the permittivity and permeability of the physical medium in which the wire is located.

The speed of light in free space is $v = c = 3 \times 10^8$ m/s. Signals in a PCB travel at about half this speed, because the FR4 insulator has four times the permittivity of air. Thus, PCB signals travel at about 1.5×10^8 m/s, or 15 cm/ns. The time delay for a signal to travel along a transmission line of length l is

$$t_d = \frac{l}{v} \tag{A.4}$$

The *characteristic impedance* of a transmission line, Z_0 (pronounced "Z-naught"), is the ratio of voltage to current in a wave traveling along the line: $Z_0 = V/I$. It is *not* the resistance of the wire (a good transmission line in a digital system typically has negligible resistance). Z_0 depends on the inductance and capacitance of the line (see the derivation in Section A.8.7) and typically has a value of 50 to 75 Ω.

$$Z_0 = \sqrt{\frac{L}{C}} \tag{A.5}$$

Figure A.15 shows the symbol for a transmission line. The symbol resembles a *coaxial cable* with an inner signal conductor and an outer grounded conductor like that used in television cable wiring.

The key to understanding the behavior of transmission lines is to visualize the wave of voltage propagating along the line at the speed of light. When the wave reaches the end of the line, it may be absorbed or reflected, depending on the termination or load at the end. Reflections travel back along the line, adding to the voltage already on the line. Terminations are classified as matched, open, short, or mismatched. The following sections explore how a wave propagates along the line and what happens to the wave when it reaches the termination.

A.8.1 Matched Termination

Figure A.16 shows a transmission line of length l with a *matched termination*, which means that the load impedance, Z_L, is equal to the characteristic impedance, Z_0. The transmission line has a characteristic impedance of 50 Ω. One end of the line is connected to a voltage source through a

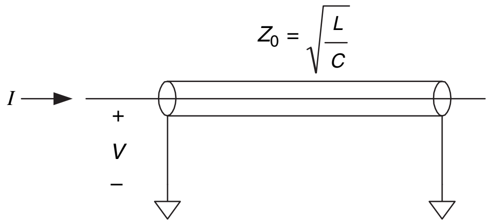


Figure A.15 **Transmission line symbol**

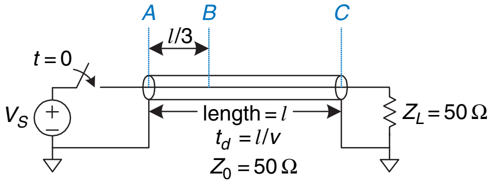


Figure A.16 Transmission line with matched termination

switch that closes at time $t = 0$. The other end is connected to the 50 Ω matched load. This section analyzes the voltages and currents at points *A*, *B*, and *C*—at the beginning of the line, one-third of the length along the line, and at the end of the line, respectively.

Figure A.17 shows the voltages at points *A*, *B*, and C over time. Initially, there is no voltage or current flowing in the transmission line, because the switch is open. At time $t = 0$, the switch closes, and the voltage source launches a wave with voltage $V = V_S$ along the line. This is called the *incident wave*. Because the characteristic impedance is Z_0, the wave has current $I = V_S/Z_0$. The voltage reaches the beginning of the line (point *A*) immediately, as shown in Figure A.17(a). The wave propagates along the line at the speed of light. At time $t_d/3$, the wave reaches point *B*. The voltage at this point abruptly rises from 0 to V_S, as shown in Figure A.17(b). At time t_d, the incident wave reaches point C at the end of the line, and the voltage rises there too. All of the current, I, flows into the resistor, Z_L, producing a voltage across the resistor of $Z_L I = Z_L (V_S/Z_0) = V_S$ because $Z_L = Z_0$. This voltage is consistent with the wave flowing along the transmission line. Thus, the wave is *absorbed* by the load impedance, and the transmission line reaches its *steady state*.

In steady state, the transmission line behaves like an ideal equipotential wire because it is, after all, just a wire. The voltage at all points along the line must be identical. Figure A.18 shows the steady-state equivalent model of the circuit in Figure A.16. The voltage is V_S everywhere along the wire.

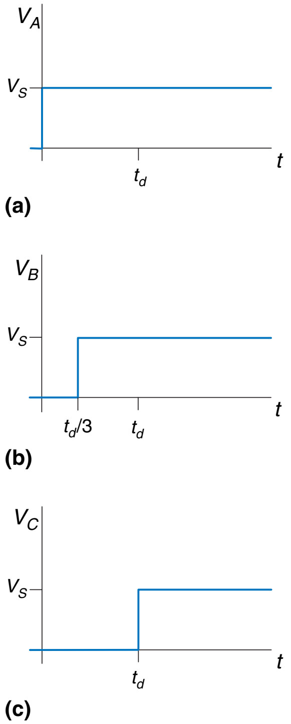


Figure A.17 Voltage waveforms for Figure A.16 at points *A*, *B*, and *C*

Example A.2 TRANSMISSION LINE WITH MATCHED SOURCE AND LOAD TERMINATIONS

Figure A.19 shows a transmission line with matched source and load impedances Z_S and Z_L. Plot the voltage at nodes *A*, *B*, and *C* versus time. When does the system reach steady-state, and what is the equivalent circuit at steady-state?

Solution: When the voltage source has a source impedance Z_S in series with the transmission line, part of the voltage drops across Z_S, and the remainder propagates down the transmission line. At first, the transmission line behaves as an impedance

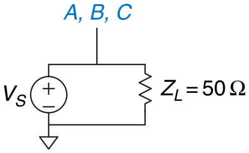


Figure A.18 Equivalent circuit of Figure A.16 at steady state

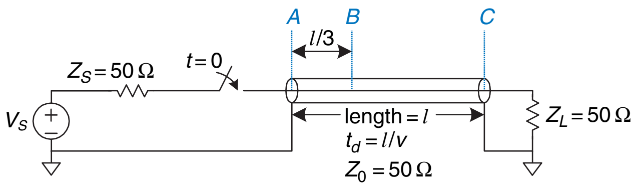


Figure A.19 Transmission line with matched source and load impedances

Z_0, because the load at the end of the line cannot possibly influence the behavior of the line until a speed of light delay has elapsed. Hence, by the *voltage divider equation*, the incident voltage flowing down the line is

$$V = V_S\left(\frac{Z_0}{Z_0 + Z_S}\right) = \frac{V_S}{2} \tag{A.6}$$

Thus, at $t = 0$, a wave of voltage, $V = \frac{V_S}{2}$, is sent down the line from point A. Again, the signal reaches point B at time $t_d/3$ and point C at t_d, as shown in Figure A.20. All of the current is absorbed by the load impedance Z_L, so the circuit enters steady-state at $t = t_d$. In steady-state, the entire line is at $V_S/2$, just as the steady-state equivalent circuit in Figure A.21 would predict.

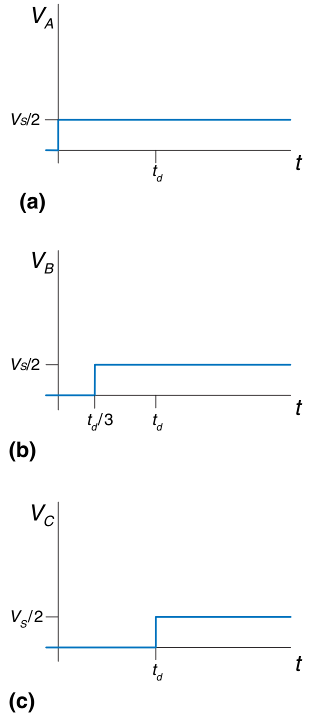


Figure A.20 Voltage waveforms for Figure A.19 at points *A*, *B*, and *C*

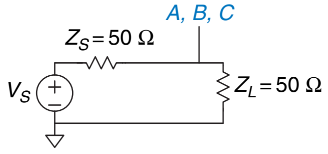


Figure A.21 Equivalent circuit of Figure A.19 at steady state

---

A.8.2 Open Termination

When the load impedance is not equal to Z_0, the termination cannot absorb all of the current, and some of the wave must be reflected. Figure A.22 shows a transmission line with an open load termination. No current can flow through an open termination, so the current at point C must always be 0.

The voltage on the line is initially zero. At $t = 0$, the switch closes and a wave of voltage, $V = V_S\frac{Z_0}{Z_0+Z_S} = \frac{V_S}{2}$, begins propagating down the line. Notice that this initial wave is the same as that of Example A.2 and is independent of the termination, because the load at the end of the line cannot influence the behavior at the beginning until at least $2t_d$ has elapsed. This wave reaches point B at $t_d/3$ and point C at t_d as shown in Figure A.23.

When the incident wave reaches point C, it cannot continue forward because the wire is open. It must instead reflect back toward the source. The reflected wave also has voltage $V = \frac{V_S}{2}$, because the open termination reflects the entire wave.

The voltage at any point is the sum of the incident and reflected waves. At time $t = t_d$, the voltage at point C is $V = \frac{V_S}{2} + \frac{V_S}{2} = V_S$. The reflected wave reaches point B at $5t_d/3$ and point A at $2t_d$. When it reaches point A,

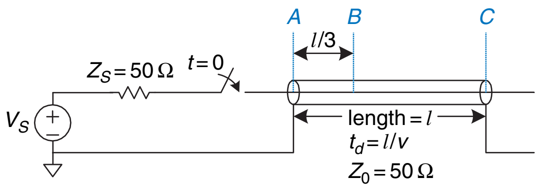


Figure A.22 Transmission line with open load termination

the wave is absorbed by the source termination impedance that matches the characteristic impedance of the line. Thus, the system reaches steady state at time $t = 2t_d$, and the transmission line becomes equivalent to an equipotential wire with voltage V_S and current $I = 0$.

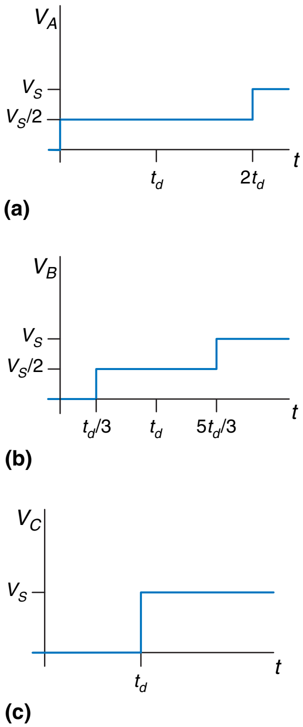


Figure A.23 Voltage waveforms for Figure A.22 at points *A*, *B*, and *C*

A.8.3 Short Termination

Figure A.24 shows a transmission line terminated with a short circuit to ground. Thus, the voltage at point C must always be 0.

As in the previous examples, the voltages on the line are initially 0. When the switch closes, a wave of voltage, $V = \frac{V_S}{2}$, begins propagating down the line (Figure A.25). When it reaches the end of the line, it must reflect with opposite polarity. The reflected wave, with voltage $V = \frac{-V_S}{2}$, adds to the incident wave, ensuring that the voltage at point C remains 0. The reflected wave reaches the source at time $t = 2t_d$ and is absorbed by the source impedance. At this point, the system reaches steady state, and the transmission line is equivalent to an equipotential wire with voltage $V = 0$.

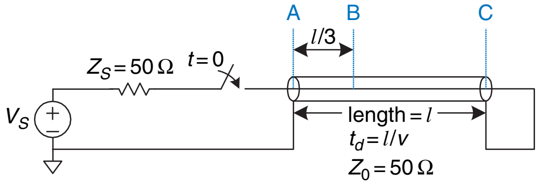


Figure A.24 Transmission line with short termination

A.8.4 Mismatched Termination

The termination impedance is said to be *mismatched* when it does not equal the characteristic impedance of the line. In general, when an incident wave reaches a mismatched termination, part of the wave is absorbed and part is reflected. The reflection coefficient k_r indicates the fraction of the incident wave V_i that is reflected: $V_r = k_r V_i$.

Section A.8.8 derives the reflection coefficient using conservation of current arguments. It shows that, when an incident wave flowing along

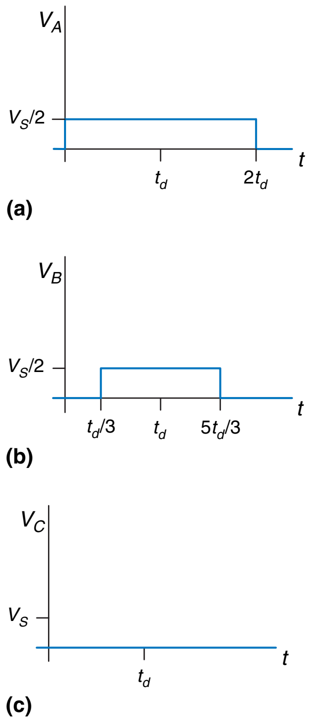


Figure A.25 **Voltage waveforms for Figure A.24 at points *A*, *B*, and *C***

a transmission line of characteristic impedance Z_0 reaches a termination impedance Z_T at the end of the line, the reflection coefficient is

$$k_r = \frac{Z_T - Z_0}{Z_T + Z_0} \tag{A.7}$$

Note a few special cases. If the termination is an open circuit ($Z_T = \infty$), $k_r = 1$, because the incident wave is entirely reflected (so the current out the end of the line remains zero). If the termination is a short circuit ($Z_T = 0$), $k_r = -1$, because the incident wave is reflected with negative polarity (so the voltage at the end of the line remains zero). If the termination is a matched load ($Z_T = Z_0$), $k_r = 0$, because the incident wave is completely absorbed.

Figure A.26 illustrates reflections in a transmission line with a *mismatched load termination* of 75 Ω. $Z_T = Z_L = 75\ \Omega$, and $Z_0 = 50\ \Omega$, so $k_r = 1/5$. As in previous examples, the voltage on the line is initially 0. When the switch closes, a wave of voltage $V = \frac{V_S}{2}$ propagates down the line, reaching the end at $t = t_d$. When the incident wave reaches the termination at the end of the line, one fifth of the wave is reflected, and the remaining four fifths flows into the load impedance. Thus, the reflected wave has a voltage $V = \frac{V_S}{2} \times \frac{1}{5} = \frac{V_S}{10}$. The total voltage at point C is the sum of the incoming and reflected voltages, $V_C = \frac{V_S}{2} + \frac{V_S}{10} = \frac{3V_S}{5}$. At $t = 2t_d$, the reflected wave reaches point A, where it is absorbed by the matched 50 Ω termination, Z_S. Figure A.27 plots the voltages and currents along the line. Again, note that, in steady state (in this case at time $t > 2t_d$), the transmission line is equivalent to an equipotential wire, as shown in Figure A.28. At steady state, the system acts like a voltage divider, so

$$V_A = V_B = V_C = V_S\left(\frac{Z_L}{Z_L + Z_S}\right) = V_S\left(\frac{75\Omega}{75\Omega + 50\Omega}\right) = \frac{3V_S}{5}$$

Reflections can occur at both ends of the transmission line. Figure A.29 shows a transmission line with a source impedance, Z_S, of 450 Ω and an open termination at the load. The reflection coefficients at the load and source, k_{rL} and k_{rS}, are 1 and 4/5, respectively. In this case, waves reflect off both ends of the transmission line until a steady state is reached.

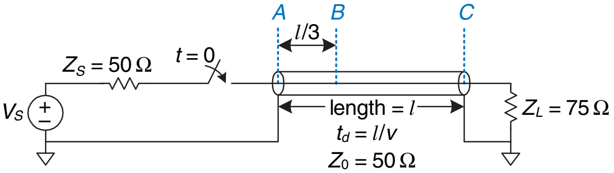


Figure A.26 **Transmission line with mismatched termination**

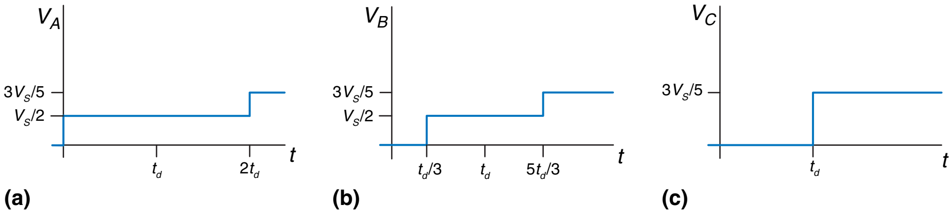


Figure A.27 Voltage waveforms for Figure A.26 at points *A*, *B*, and *C*

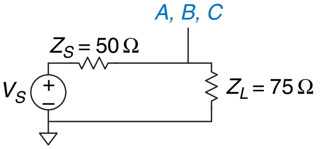

A, B, C
$Z_S = 50\,\Omega$
V_S
$Z_L = 75\,\Omega$

Figure A.28 Equivalent circuit of Figure A.26 at steady state

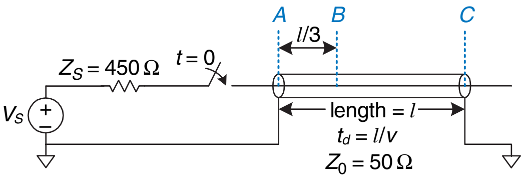

A
B
C
$l/3$
$Z_S = 450\,\Omega$
$t = 0$
V_S
length = l
$t_d = l/v$
$Z_0 = 50\,\Omega$

Figure A.29 Transmission line with mismatched source and load terminations

The *bounce diagram* shown in Figure A.30 helps visualize reflections off both ends of the transmission line. The horizontal axis represents distance along the transmission line, and the vertical axis represents time, increasing downward. The two sides of the bounce diagram represent the source and load ends of the transmission line, points *A* and *C*. The incoming and reflected signal waves are drawn as diagonal lines between points *A* and *C*. At time $t = 0$, the source impedance and transmission line behave as a voltage divider, launching a voltage wave of $\frac{V_S}{10}$ from point *A* toward point *C*. At time $t = t_d$, the signal reaches point *C* and is completely reflected ($k_{rL} = 1$). At time $t = 2t_d$, the reflected wave of $\frac{V_S}{10}$ reaches point *A* and is reflected with a reflection coefficient, $k_{rS} = 4/5$, to produce a wave of $\frac{2V_S}{25}$ traveling toward point *C*, and so forth.

The voltage at a given time at any point on the transmission line is the sum of all the incident and reflected waves. Thus, at time $t = 1.1t_d$, the voltage at point C is $\frac{V_S}{10} + \frac{V_S}{10} = \frac{V_S}{5}$. At time $t = 3.1t_d$, the voltage at point C is $\frac{V_S}{10} + \frac{V_S}{10} + \frac{2V_S}{25} + \frac{2V_S}{25} = \frac{9V_S}{25}$, and so forth. Figure A.31 plots the voltages

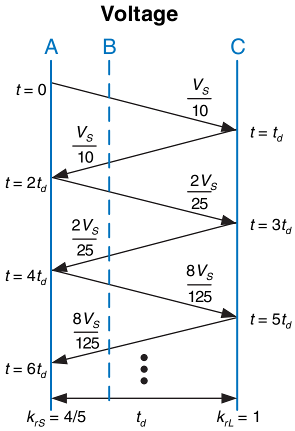

Voltage
A
B
C
$t = 0$
$\frac{V_S}{10}$
$t = t_d$
$\frac{V_S}{10}$
$t = 2t_d$
$\frac{2V_S}{25}$
$t = 3t_d$
$\frac{2V_S}{25}$
$t = 4t_d$
$\frac{8V_S}{125}$
$t = 5t_d$
$\frac{8V_S}{125}$
$t = 6t_d$
$k_{rS} = 4/5$
t_d
$k_{rL} = 1$

Figure A.30 Bounce diagram for Figure A.29

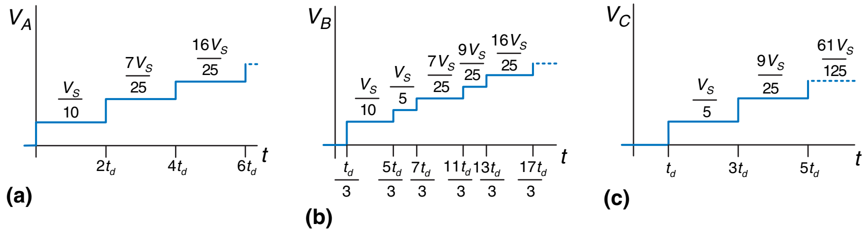


Figure A.31 Voltage and current waveforms for Figure A.29

against time. As t approaches infinity, the voltages approach steady state with $V_A = V_B = V_C = V_S$.

A.8.5 When to Use Transmission Line Models

Transmission line models for wires are needed whenever the wire delay, t_d, is longer than a fraction (e.g., 20%) of the edge rates (rise or fall times) of a signal. If the wire delay is shorter, it has an insignificant effect on the propagation delay of the signal, and the reflections dissipate while the signal is transitioning. If the wire delay is longer, it must be considered in order to accurately predict the propagation delay and waveform of the signal. In particular, reflections may distort the digital characteristic of a waveform, resulting in incorrect logic operations.

Recall that signals travel on a PCB at about 15 cm/ns. For TTL logic, with edge rates of 10 ns, wires must be modeled as transmission lines only if they are longer than 30 cm (10 ns × 15 cm/ns × 20%). PCB traces are usually less than 30 cm, so most traces can be modeled as ideal equipotential wires. In contrast, many modern chips have edge rates of 2 ns or less, so traces longer than about 6 cm (about 2.5 inches) must be modeled as transmission lines. Clearly, use of edge rates that are crisper than necessary just causes difficulties for the designer.

Breadboards lack a ground plane, so the electromagnetic fields of each signal are nonuniform and difficult to model. Moreover, the fields interact with other signals. This can cause strange reflections and crosstalk between signals. Thus, breadboards are unreliable above a few megahertz.

In contrast, PCBs have good transmission lines with consistent characteristic impedance and velocity along the entire line. As long as they are terminated with a source or load impedance that is matched to the impedance of the line, PCB traces do not suffer from reflections.

A.8.6 Proper Transmission Line Terminations

There are two common ways to properly terminate a transmission line, shown in Figure A.32. In *parallel termination*, the driver has a low impedance ($Z_S \approx 0$). A load resistor Z_L with impedance Z_0 is placed in parallel

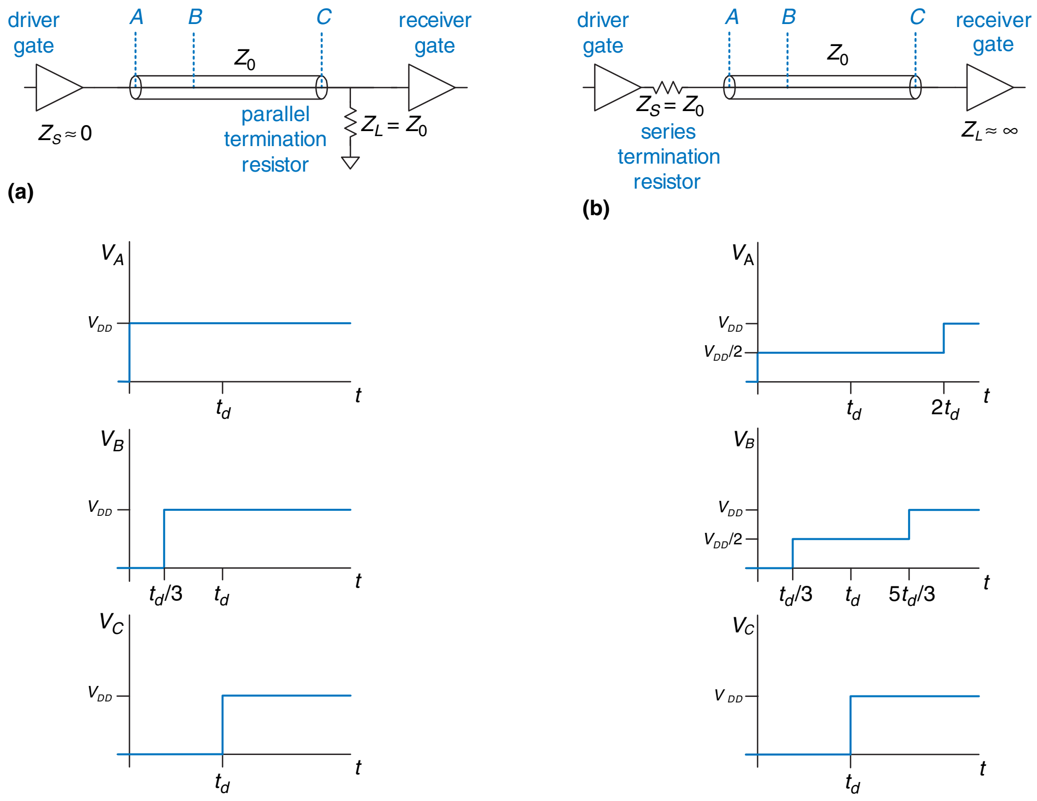

driver gate
A
B
C
Z_0
receiver gate
$Z_S \approx 0$
parallel termination resistor
$Z_L = Z_0$
(a)
driver gate
A
B
C
Z_0
receiver gate
$Z_S = Z_0$
series termination resistor
$Z_L \approx \infty$
(b)
V_A
V_{DD}
t_d
t
V_B
V_{DD}
$t_d/3$
t_d
t
V_C
V_{DD}
t_d
t
V_A
V_{DD}
$V_{DD}/2$
t_d
$2t_d$
t
V_B
V_{DD}
$V_{DD}/2$
$t_d/3$
t_d
$5t_d/3$
t
V_C
V_{DD}
t_d
t

Figure A.32 Termination schemes: (a) parallel, (b) series

with the load (between the input of the receiver gate and ground). When the driver switches from 0 to V_{DD}, it sends a wave with voltage V_{DD} down the line. The wave is absorbed by the matched load termination, and no reflections take place. In *series termination*, a source resistor Z_S is placed in series with the driver to raise the source impedance to Z_0. The load has a high impedance ($Z_L \approx \infty$). When the driver switches, it sends a wave with voltage $V_{DD}/2$ down the line. The wave reflects at the open circuit load and returns, bringing the voltage on the line up to V_{DD}. The wave is absorbed at the source termination. Both schemes are similar in that the voltage at the receiver transitions from 0 to V_{DD} at $t = t_d$, just as one would desire. They differ in power consumption and in the waveforms that appear elsewhere along the line. Parallel termination dissipates power continuously through the load resistor when the line is at a high voltage. Series termination dissipates no DC power, because the load is an open circuit. However, in series terminated lines, points near the middle of the transmission line initially see a voltage of $V_{DD}/2$, until the reflection returns. If other gates are attached to the middle of the line, they will momentarily see an illegal logic level. Therefore, series termination works best for *point-to-point* communication with a single driver and a single receiver. Parallel termination is better for a *bus* with multiple receivers, because receivers at the middle of the line never see an illegal logic level.

A.8.7 Derivation of Z_0*

Z_0 is the ratio of voltage to current in a wave propagating along a transmission line. This section derives Z_0; it assumes some previous knowledge of resistor-inductor-capacitor (RLC) circuit analysis.

Imagine applying a step voltage to the input of a semi-infinite transmission line (so that there are no reflections). Figure A.33 shows the semi-infinite line and a model of a segment of the line of length dx. R, L, and C, are the values of resistance, inductance, and capacitance per unit length. Figure A.33(b) shows the transmission line model with a resistive component, R. This is called a *lossy* transmission line model, because energy is dissipated, or lost, in the resistance of the wire. However, this loss is often negligible, and we can simplify analysis by ignoring the resistive component and treating the transmission line as an *ideal* transmission line, as shown in Figure A.33(c).

Voltage and current are functions of time and space throughout the transmission line, as given by Equations A.8 and A.9.

$$\frac{\partial}{\partial x}V(x,t) = L\frac{\partial}{\partial t}I(x,t) \tag{A.8}$$

$$\frac{\partial}{\partial x}I(x,t) = C\frac{\partial}{\partial t}V(x,t) \tag{A.9}$$

Taking the space derivative of Equation A.8 and the time derivative of Equation A.9 and substituting gives Equation A.10, the *wave equation*.

$$\frac{\partial^2}{\partial x^2}V(x,t) = LC\frac{\partial^2}{\partial t^2}V(x,t) \tag{A.10}$$

Z_0 is the ratio of voltage to current in the transmission line, as illustrated in Figure A.34(a). Z_0 must be independent of the length of the line, because the behavior of the wave cannot depend on things at a distance. Because it is independent of length, the impedance must still equal Z_0 after the addition of a small amount of transmission line, dx, as shown in Figure A.34(b).

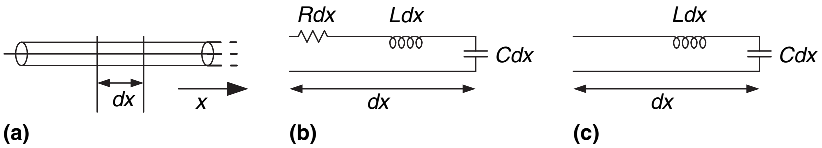


Figure A.33 Transmission line models: (a) semi-infinite cable, (b) lossy, (c) ideal

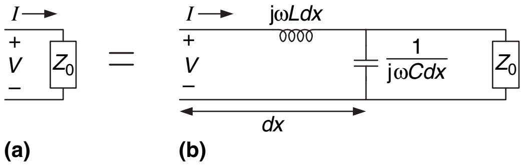


Figure A.34 Transmission line model: (a) for entire line and (b) with additional length, *dx*

Using the impedances of an inductor and a capacitor, we rewrite the relationship of Figure A.34 in equation form:

$$Z_0 = j\omega L dx + [Z_0 || (1/(j\omega C dx))] \tag{A.11}$$

Rearranging, we get

$$Z_0^2(j\omega C) - j\omega L + \omega^2 Z_0 L C dx = 0 \tag{A.12}$$

Taking the limit as dx approaches 0, the last term vanishes and we find that

$$Z_0 = \sqrt{\frac{L}{C}} \tag{A.13}$$

A.8.8 Derivation of the Reflection Coefficient*

The reflection coefficient k_r is derived using conservation of current. Figure A.35 shows a transmission line with characteristic impedance Z_0 and load impedance Z_L. Imagine an incident wave of voltage V_i and current I_i. When the wave reaches the termination, some current I_L flows through the load impedance, causing a voltage drop V_L. The remainder of the current reflects back down the line in a wave of voltage V_r and current I_r. Z_0 is the ratio of voltage to current in waves propagating along the line, so $\frac{V_i}{I_i} = \frac{V_r}{I_r} = Z_0$.

The voltage on the line is the sum of the voltages of the incident and reflected waves. The current flowing in the positive direction on the line is the difference between the currents of the incident and reflected waves.

$$V_L = V_i + V_r \tag{A.14}$$

$$I_L = I_i – I_r \tag{A.15}$$

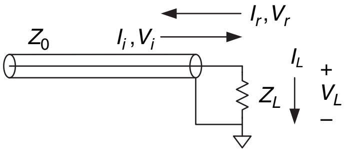


Figure A.35 Transmission line showing incoming, reflected, and load voltages and currents

Using Ohm's law and substituting for I_L, I_i, and I_r in Equation A.15, we get

$$\frac{V_i + V_r}{Z_L} = \frac{V_i}{Z_0} - \frac{V_r}{Z_0} \tag{A.16}$$

Rearranging, we solve for the reflection coefficient, k_r:

$$\frac{V_r}{V_i} = \frac{Z_L - Z_0}{Z_L + Z_0} = k_r \tag{A.17}$$

A.8.9 Putting It All Together

Transmission lines model the fact that signals take time to propagate down long wires because the speed of light is finite. An ideal transmission line has uniform inductance L and capacitance C per unit length and zero resistance. The transmission line is characterized by its characteristic impedance Z_0 and delay t_d which can be derived from the inductance, capacitance, and wire length. The transmission line has significant delay and noise effects on signals whose rise/fall times are less than about $5t_d$. This means that, for systems with 2 ns rise/fall times, PCB traces longer than about 6 cm must be analyzed as transmission lines to accurately understand their behavior.

A digital system consisting of a gate driving a long wire attached to the input of a second gate can be modeled with a transmission line as shown in Figure A.36. The voltage source, source impedance Z_S, and switch model the first gate switching from 0 to 1 at time 0. The driver gate cannot supply infinite current; this is modeled by Z_S. Z_S is usually small for a logic gate, but a designer may choose to add a resistor in series with the gate to raise Z_S and match the impedance of the line. The input to the second gate is modeled as Z_L. CMOS circuits usually have little input current, so Z_L may be close to infinity. The designer may also choose to add a resistor in parallel with the second gate, between the gate input and ground, so that Z_L matches the impedance of the line.

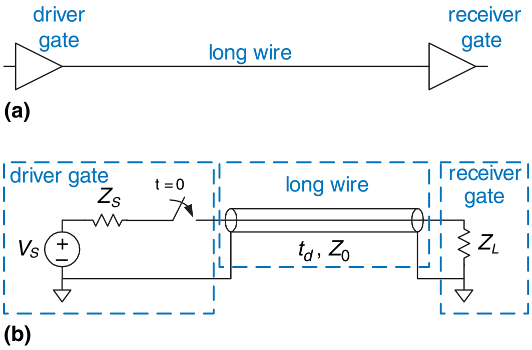


Figure A.36 Digital system modeled with transmission line

When the first gate switches, a wave of voltage is driven onto the transmission line. The source impedance and transmission line form a voltage divider, so the voltage of the incident wave is

$$V_i = V_S \frac{Z_0}{Z_0 + Z_S} \tag{A.18}$$

At time t_d, the wave reaches the end of the line. Part is absorbed by the load impedance, and part is reflected. The reflection coefficient k_r indicates the portion that is reflected: $k_r = V_r/V_i$, where V_r is the voltage of the reflected wave and V_i is the voltage of the incident wave.

$$k_r = \frac{Z_L - Z_0}{Z_L + Z_0} \tag{A.19}$$

The reflected wave adds to the voltage already on the line. It reaches the source at time $2t_d$, where part is absorbed and part is again reflected. The reflections continue back and forth, and the voltage on the line eventually approaches the value that would be expected if the line were a simple equipotential wire.

A.9 ECONOMICS

Although digital design is so much fun that some of us would do it for free, most designers and companies intend to make money. Therefore, economic considerations are a major factor in design decisions.

The cost of a digital system can be divided into *nonrecurring engineering costs* (*NRE*), and *recurring costs*. NRE accounts for the cost of designing the system. It includes the salaries of the design team, computer and software costs, and the costs of producing the first working unit. The fully loaded cost of a designer in the United States in 2012 (including salary, health insurance, retirement plan, and a computer with design tools) was roughly $200,000 per year, so design costs can be significant. Recurring costs are the cost of each additional unit; this includes components, manufacturing, marketing, technical support, and shipping.

The sales price must cover not only the cost of the system but also other costs such as office rental, taxes, and salaries of staff who do not directly contribute to the design (such as the janitor and the CEO). After all of these expenses, the company should still make a profit.

Example A.3 BEN TRIES TO MAKE SOME MONEY

Ben Bitdiddle has designed a crafty circuit for counting raindrops. He decides to sell the device and try to make some money, but he needs help deciding what implementation to use. He decides to use either an FPGA or an ASIC. The development kit to design and test the FPGA costs \$1500. Each FPGA costs \$17. The ASIC costs \$600,000 for a mask set and \$4 per chip.

Regardless of what chip implementation he chooses, Ben needs to mount the packaged chip on a printed circuit board (PCB), which will cost him \$1.50 per board. He thinks he can sell 1000 devices per month. Ben has coerced a team of bright undergraduates into designing the chip for their senior project, so it doesn't cost him anything to design.

If the sales price has to be twice the cost (100% profit margin), and the product life is 2 years, which implementation is the better choice?

Solution: Ben figures out the total cost for each implementation over 2 years, as shown in Table A.4. Over 2 years, Ben plans on selling 24,000 devices, and the total cost is given in Table A.4 for each option. If the product life is only two years, the FPGA option is clearly superior. The per-unit cost is \$445,500/24,000 = \$18.56, and the sales price is \$37.13 per unit to give a 100% profit margin. The ASIC option would have cost \$732,000/24,000 = \$30.50 and would have sold for \$61 per unit.

Table A.4 **ASIC vs FPGA costs**

Cost	ASIC	FPGA
NRE	\$600,000	\$1500
chip	\$4	\$17
PCB	\$1.50	\$1.50
TOTAL	\$600,000 + (24,000 × \$5.50) = **\$732,000**	\$1500 + (24,000 × \$18.50) = **\$445,500**
per unit	\$30.50	\$18.56

Example A.4 BEN GETS GREEDY

After seeing the marketing ads for his product, Ben thinks he can sell even more chips per month than originally expected. If he were to choose the ASIC option, how many devices per month would he have to sell to make the ASIC option more profitable than the FPGA option?

Solution: Ben solves for the minimum number of units, N, that he would need to sell in 2 years:

$$\$600,000 + (N \times \$5.50) = \$1500 + (N \times \$18.50)$$

Solving the equation gives $N = 46{,}039$ units, or 1919 units per month. He would need to almost double his monthly sales to benefit from the ASIC solution.

Example A.5 BEN GETS LESS GREEDY

Ben realizes that his eyes have gotten too big for his stomach, and he doesn't think he can sell more than 1000 devices per month. But he does think the product life can be longer than 2 years. At a sales volume of 1000 devices per month, how long would the product life have to be to make the ASIC option worthwhile?

Solution: If Ben sells more than 46,039 units in total, the ASIC option is the best choice. So, Ben would need to sell at a volume of 1000 per month for at least 47 months (rounding up), which is almost 4 years. By then, his product is likely to be obsolete.

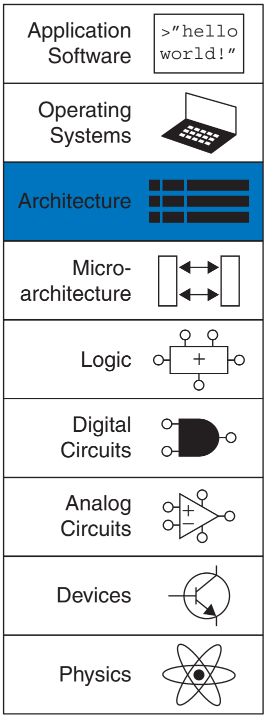
Chips are usually purchased from a distributor rather than directly from the manufacturer (unless you are ordering tens of thousands of units). Digikey (`www.digikey.com`) is a leading distributor that sells a wide variety of electronics. Jameco (`www.jameco.com`) and All Electronics (`www.allelectronics.com`) have eclectic catalogs that are competitively priced and well suited to hobbyists.

This page intentionally left blank

MIPS Instructions

B

This appendix summarizes MIPS instructions used in this book. Tables B.1–B.3 define the `opcode` and `funct` fields for each instruction, along with a short description of what the instruction does. The following notations are used:


Application Software
>"hello world!"
Operating Systems
Architecture
Micro-architecture
Logic
+
Digital Circuits
Analog Circuits
+
−
Devices
Physics

- `[reg]`: contents of the register
- `imm`: 16-bit immediate field of the I-type instruction
- `addr`: 26-bit address field of the J-type instruction
- `SignImm`: 32-bit sign-extended immediate

  `= {{16{imm[15]}}, imm}`
- `ZeroImm`: 32-bit zero-extended immediate

  `= {16'b0, imm}`
- `Address`: `[rs] + SignImm`
- `[Address]`: contents of memory location `Address`
- `BTA`: branch target address[1]

  `= PC + 4 + (SignImm << 2)`
- `JTA`: jump target address

  `= {(PC + 4)[31:28], addr, 2'b0}`
- `label`: text indicating an instruction location

---

[1] The BTA in the SPIM simulator is PC + (SignImm << 2) because it has no branch delay slot. Thus, if you use the SPIM assembler to create machine code for a real MIPS processor, you must decrement the immediate field of each branch instruction by 1 to compensate.

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00010-0
© 2013 Elsevier, Inc. All rights reserved.

Table B.1 Instructions, sorted by opcode

Opcode	Name	Description	Operation
000000 (0)	R-type	all R-type instructions	see Table B.2
000001 (1) (rt = 0/1)	bltz rs, label / bgez rs, label	branch less than zero/branch greater than or equal to zero	if ([rs] < 0) PC = BTA/ if ([rs] ≥ 0) PC = BTA
000010 (2)	j label	jump	PC = JTA
000011 (3)	jal label	jump and link	$ra = PC + 4, PC = JTA
000100 (4)	beq rs, rt, label	branch if equal	if ([rs] == [rt]) PC = BTA
000101 (5)	bne rs, rt, label	branch if not equal	if ([rs] != [rt]) PC = BTA
000110 (6)	blez rs, label	branch if less than or equal to zero	if ([rs] ≤ 0) PC = BTA
000111 (7)	bgtz rs, label	branch if greater than zero	if ([rs] > 0) PC = BTA
001000 (8)	addi rt, rs, imm	add immediate	[rt] = [rs] + SignImm
001001 (9)	addiu rt, rs, imm	add immediate unsigned	[rt] = [rs] + SignImm
001010 (10)	slti rt, rs, imm	set less than immediate	[rs] < SignImm ? [rt] = 1 : [rt] = 0
001011 (11)	sltiu rt, rs, imm	set less than immediate unsigned	[rs] < SignImm ? [rt] = 1 : [rt] = 0
001100 (12)	andi rt, rs, imm	and immediate	[rt] = [rs] & ZeroImm
001101 (13)	ori rt, rs, imm	or immediate	[rt] = [rs] | ZeroImm
001110 (14)	xori rt, rs, imm	xor immediate	[rt] = [rs] ^ ZeroImm
001111 (15)	lui rt, imm	load upper immediate	[rt] = {imm, 16'b0}
010000 (16) (rs = 0/4)	mfc0 rt, rd / mtc0 rt, rd	move from/to coprocessor 0	[rt] = [rd]/[rd] = [rt] (rd is in coprocessor 0)
010001 (17)	F-type	fop = 16/17: F-type instructions	see Table B.3
010001 (17) (rt = 0/1)	bc1f label/ bc1t label	fop = 8: branch if fpcond is FALSE/TRUE	if (fpcond == 0) PC = BTA/ if (fpcond == 1) PC = BTA
011100 (28) (func = 2)	mul rd, rs, rt	multiply (32-bit result)	[rd] = [rs] x [rt]
100000 (32)	lb rt, imm(rs)	load byte	[rt] = SignExt ([Address]$_{7:0}$)
100001 (33)	lh rt, imm(rs)	load halfword	[rt] = SignExt ([Address]$_{15:0}$)
100011 (35)	lw rt, imm(rs)	load word	[rt] = [Address]
100100 (36)	lbu rt, imm(rs)	load byte unsigned	[rt] = ZeroExt ([Address]$_{7:0}$)
100101 (37)	lhu rt, imm(rs)	load halfword unsigned	[rt] = ZeroExt ([Address]$_{15:0}$)

(continued)

Table B.1 **Instructions, sorted by opcode—Cont'd**

Opcode	Name	Description	Operation
101000 (40)	sb rt, imm(rs)	store byte	$[\text{Address}]_{7:0} = [\text{rt}]_{7:0}$
101001 (41)	sh rt, imm(rs)	store halfword	$[\text{Address}]_{15:0} = [\text{rt}]_{15:0}$
101011 (43)	sw rt, imm(rs)	store word	[Address] = [rt]
110001 (49)	lwc1 ft, imm(rs)	load word to FP coprocessor 1	[ft] = [Address]
111001 (56)	swc1 ft, imm(rs)	store word to FP coprocessor 1	[Address] = [ft]

Table B.2 **R-type instructions, sorted by funct field**

Funct	Name	Description	Operation
000000 (0)	sll rd, rt, shamt	shift left logical	[rd] = [rt] << shamt
000010 (2)	srl rd, rt, shamt	shift right logical	[rd] = [rt] >> shamt
000011 (3)	sra rd, rt, shamt	shift right arithmetic	[rd] = [rt] >>> shamt
000100 (4)	sllv rd, rt, rs	shift left logical variable	$[\text{rd}] = [\text{rt}] \ll [\text{rs}]_{4:0}$
000110 (6)	srlv rd, rt, rs	shift right logical variable	$[\text{rd}] = [\text{rt}] \gg [\text{rs}]_{4:0}$
000111 (7)	srav rd, rt, rs	shift right arithmetic variable	$[\text{rd}] = [\text{rt}] \ggg [\text{rs}]_{4:0}$
001000 (8)	jr rs	jump register	PC = [rs]
001001 (9)	jalr rs	jump and link register	$ra = PC + 4, PC = [rs]
001100 (12)	syscall	system call	system call exception
001101 (13)	break	break	break exception
010000 (16)	mfhi rd	move from hi	[rd] = [hi]
010001 (17)	mthi rs	move to hi	[hi] = [rs]
010010 (18)	mflo rd	move from lo	[rd] = [lo]
010011 (19)	mtlo rs	move to lo	[lo] = [rs]
011000 (24)	mult rs, rt	multiply	{[hi], [lo]} = [rs] × [rt]
011001 (25)	multu rs, rt	multiply unsigned	{[hi], [lo]} = [rs] × [rt]
011010 (26)	div rs, rt	divide	[lo] = [rs]/[rt], [hi] = [rs]%[rt]
011011 (27)	divu rs, rt	divide unsigned	[lo] = [rs]/[rt], [hi] = [rs]%[rt]

(*continued*)

Table B.2 R-type instructions, sorted by funct field—Cont'd

Funct	Name	Description	Operation
100000 (32)	`add rd, rs, rt`	add	`[rd] = [rs] + [rt]`
100001 (33)	`addu rd, rs, rt`	add unsigned	`[rd] = [rs] + [rt]`
100010 (34)	`sub rd, rs, rt`	subtract	`[rd] = [rs] - [rt]`
100011 (35)	`subu rd, rs, rt`	subtract unsigned	`[rd] = [rs] - [rt]`
100100 (36)	`and rd, rs, rt`	and	`[rd] = [rs] & [rt]`
100101 (37)	`or rd, rs, rt`	or	`[rd] = [rs] | [rt]`
100110 (38)	`xor rd, rs, rt`	xor	`[rd] = [rs] ^ [rt]`
100111 (39)	`nor rd, rs, rt`	nor	`[rd] = ~([rs] | [rt])`
101010 (42)	`slt rd, rs, rt`	set less than	`[rs] < [rt] ? [rd] = 1 : [rd] = 0`
101011 (43)	`sltu rd, rs, rt`	set less than unsigned	`[rs] < [rt] ? [rd] = 1 : [rd] = 0`

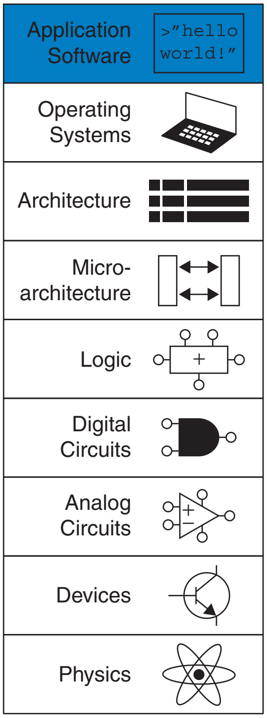
Table B.3 F-type instructions (`fop` = 16/17)

Funct	Name	Description	Operation
000000 (0)	`add.s fd, fs, ft /` `add.d fd, fs, ft`	FP add	`[fd] = [fs] + [ft]`
000001 (1)	`sub.s fd, fs, ft /` `sub.d fd, fs, ft`	FP subtract	`[fd] = [fs] - [ft]`
000010 (2)	`mul.s fd, fs, ft /` `mul.d fd, fs, ft`	FP multiply	`[fd] = [fs] × [ft]`
000011 (3)	`div.s fd, fs, ft /` `div.d fd, fs, ft`	FP divide	`[fd] = [fs]/[ft]`
000101 (5)	`abs.s fd, fs /` `abs.d fd, fs`	FP absolute value	`[fd] = ([fs] < 0) ? [-fs]` `: [fs]`
000111 (7)	`neg.s fd, fs /` `neg.d fd, fs`	FP negation	`[fd] = [-fs]`
111010 (58)	`c.seq.s fs, ft /` `c.seq.d fs, ft`	FP equality comparison	`fpcond = ([fs] == [ft])`
111100 (60)	`c.lt.s fs, ft /` `c.lt.d fs, ft`	FP less than comparison	`fpcond = ([fs] < [ft])`
111110 (62)	`c.le.s fs, ft /` `c.le.d fs, ft`	FP less than or equal comparison	`fpcond = ([fs] ≤ [ft])`

C Programming

C

C.1 **Introduction**
C.2 **Welcome to C**
C.3 **Compilation**
C.4 **Variables**
C.5 **Operators**
C.6 **Function Calls**
C.7 **Control-Flow Statements**
C.8 **More Data Types**
C.9 **Standard Libraries**
C.10 **Compiler and Command Line Options**
C.11 **Common Mistakes**



C.1 INTRODUCTION

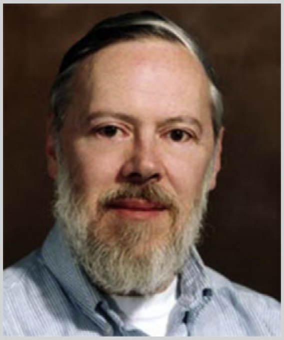
The overall goal of this book is to give a picture of how computers work on many levels, from the transistors by which they are constructed all the way up to the software they run. The first five chapters of this book work up through the lower levels of abstraction, from transistors to gates to logic design. Chapters 6 through 8 jump up to architecture and work back down to microarchitecture to connect the hardware with the software. This Appendix on C programming fits logically between Chapters 5 and 6, covering C programming as the highest level of abstraction in the text. It motivates the architecture material and links this book to programming experience that may already be familiar to the reader. This material is placed in the Appendix so that readers may easily cover or skip it depending on previous experience.

Programmers use many different languages to tell a computer what to do. Fundamentally, computers process instructions in *machine language* consisting of 1's and 0's, as will be explored in Chapter 6. But programming in machine language is tedious and slow, leading programmers to use more abstract languages to get their meaning across more efficiently. Table C.1 lists some examples of languages at various levels of abstraction.

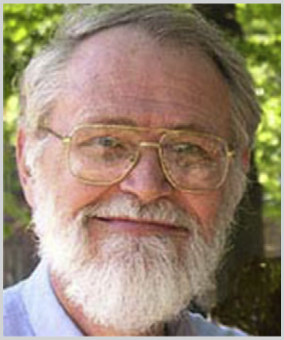
One of the most popular programming languages ever developed is called C. It was created by a group including Dennis Ritchie and Brian Kernighan at Bell Laboratories between 1969 and 1973 to rewrite the UNIX operating system from its original assembly language. By many measures, C (including a family of closely related languages such as C++, C#, and Objective C) is the most widely used language in existence. Its popularity stems from a number of factors including:

- Availability on a tremendous variety of platforms, from supercomputers down to embedded microcontrollers
- Relative ease of use, with a huge user base

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00017-3
© 2013 Elsevier, Inc. All rights reserved.



Dennis Ritchie, 1941–2011



Brian Kernighan, 1942–

C was formally introduced in 1978 by Brian Kernighan and Dennis Ritchie's classic book, *The C Programming Language*. In 1989, the American National Standards Institute (ANSI) expanded and standardized the language, which became known as ANSI C, Standard C, or C89. Shortly thereafter, in 1990, this standard was adopted by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). ISO/IEC updated the standard in 1999 to what is called C99, which we will be discussing in this text.

Table C.1 Languages at roughly decreasing levels of abstraction

Language	Description
Matlab	Designed to facilitate heavy use of math functions
Perl	Designed for scripting
Python	Designed to emphasize code readability
Java	Designed to run securely on any computer
C	Designed for flexibility and overall system access, including device drivers
Assembly Language	Human-readable machine language
Machine Language	Binary representation of a program

- Moderate level of abstraction providing higher productivity than assembly language, yet giving the programmer a good understanding of how the code will be executed
- Suitability for generating high performance programs
- Ability to interact directly with the hardware

This chapter is devoted to C programming for a variety of reasons. Most importantly, C allows the programmer to directly access addresses in memory, illustrating the connection between hardware and software emphasized in this book. C is a practical language that all engineers and computer scientists should know. Its uses in many aspects of implementation and design – e.g., software development, embedded systems programming, and simulation – make proficiency in C a vital and marketable skill.

The following sections describe the overall syntax of a C program, discussing each part of the program including the header, function and variable declarations, data types, and commonly used functions provided in libraries. Section 8.6 describes a hands-on application by using C to program a PIC32 microcontroller.

SUMMARY

- **High-level programming:** High-level programming is useful at many levels of design, from writing analysis or simulation software to programming microcontrollers that interact with hardware.

- **Low-level access:** C code is powerful because, in addition to high-level constructs, it provides access to low-level hardware and memory.

C.2 WELCOME TO C

A C program is a text file that describes operations for the computer to perform. The text file is *compiled*, converted into a machine-readable format, and run or *executed* on a computer. C Code Example C.1 is a simple C program that prints the phrase “Hello world!” to the *console*, the computer screen. C programs are generally contained in one or more text files that end in “.c”. Good programming style requires a file name that indicates the contents of the program – for example, this file could be called `hello.c`.

C is the language used to program such ubiquitous systems as Linux, Windows, and iOS. C is a powerful language because of its direct access to hardware. As compared with other high level languages, for example Perl and Matlab, C does not have as much built-in support for specialized operations such as file manipulation, pattern matching, matrix manipulation, and graphical user interfaces. It also lacks features to protect the programmer from common mistakes, such as writing data past the end of an array. Its power combined with its lack of protection has assisted hackers who exploit poorly written software to break into computer systems.

C Code Example C.1 SIMPLE C PROGRAM

```
// Write "Hello world!" to the console
#include <stdio.h>

int main(void){
  printf("Hello world!\n");
}
```

Console Output

```
Hello world!
```

C.2.1 C Program Dissection

In general, a C program is organized into one or more functions. Every program must include the `main` function, which is where the program starts executing. Most programs use other functions defined elsewhere in the C code and/or in a library. The overall sections of the `hello.c` program are the header, the `main` function, and the body.

Header: `#include <stdio.h>`
The header includes the *library functions* needed by the program. In this case, the program uses the `printf` function, which is part of the standard I/O library, `stdio.h`. See Section C.9 for further details on C’s built-in libraries.

Main function: `int main(void)`
All C programs must include exactly one `main` function. Execution of the program occurs by running the code inside `main`, called the *body* of `main`. Function syntax is described in Section C.6. The body of a function contains a sequence of *statements*. Each statement ends with a semicolon. `int` denotes that the `main` function outputs, or *returns*, an integer result that indicates whether the program ran successfully.

While this chapter provides a fundamental understanding of C programming, entire texts are written that describe C in depth. One of our favorites is the classic text *The C Programming Language* by Brian Kernighan and Dennis Ritchie, the developers of C. This text gives a concise description of the nuts and bolts of C. Another good text is *A Book on C* by Al Kelley and Ira Pohl.

Body: `printf("Hello world!\n");`
The body of this `main` function contains one statement, a call to the `printf` function, which prints the phrase "Hello world!" followed by a newline character indicated by the special sequence `"\n"`. Further details about I/O functions are described in Section C.9.1.

All programs follow the general format of the simple `hello.c` program. Of course, very complex programs may contain millions of lines of code and span hundreds of files.

C.2.2 Running a C Program

C programs can be run on many different machines. This *portability* is another advantage of C. The program is first compiled on the desired machine using the C *compiler*. Slightly different versions of the C compiler exist, including *cc* (C compiler), or *gcc* (GNU C compiler). Here we show how to compile and run a C program using gcc, which is freely available for download. It runs directly on Linux machines and is accessible under the Cygwin environment on Windows machines. It is also available for many embedded systems such as Microchip PIC32 microcontrollers. The general process described below of C file creation, compilation, and execution is the same for any C program.

1. Create the text file, for example `hello.c`.
2. In a terminal window, change to the directory that contains the file `hello.c` and type `gcc hello.c` at the command prompt.
3. The compiler creates an executable file. By default, the executable is called `a.out` (or `a.exe` on Windows machines).
4. At the command prompt, type `./a.out` (or `./a.exe` on Windows) and press return.
5. "Hello world!" will appear on the screen.

SUMMARY

- `filename.c`: C program files are typically named with a .c extension.
- `main`: Each C program must have exactly one `main` function.
- `#include`: Most C programs use functions provided by built-in libraries. These functions are used by writing `#include <library.h>` at the top of the C file.
- `gcc filename.c`: C files are converted into an executable using a compiler such as the GNU compiler (gcc) or the C compiler (cc).
- **Execution:** After compilation, C programs are executed by typing `./a.out` (or `./a.exe`) at the command line prompt.

C.3 COMPILATION

A compiler is a piece of software that reads a program in a high-level language and converts it into a file of machine code called an executable. Entire textbooks are written on compilers, but we describe them here briefly. The overall operation of the compiler is to (1) preprocess the file by including referenced libraries and expanding macro definitions, (2) ignore all unnecessary information such as comments, (3) translate the high-level code into simple instructions native to the processor that are represented in binary, called machine language, and (4) compile all the instructions into a single binary executable that can be read and executed by the computer. Each machine language is specific to a given processor, so a program must be compiled specifically for the system on which it will run. For example, the MIPS machine language is covered in Chapter 6 in detail.

C.3.1 Comments

Programmers use comments to describe code at a high-level and clarify code function. Anyone who has read uncommented code can attest to their importance. C programs use two types of comments: Single-line comments begin with `//` and terminate at the end of the line; multiple-line comments begin with `/*` and end with `*/`. While comments are critical to the organization and clarity of a program, they are ignored by the compiler.

```
// This is an example of a one-line comment.

/* This is an example
   of a multi-line comment. */
```

A comment at the top of each C file is useful to describe the file's author, creation and modification dates, and purpose. The comment below could be included at the top of the `hello.c` file.

```
// hello.c
// 1 June 2012 Sarah_Harris@hmc.edu, David_Harris@hmc.edu
//
// This program prints "Hello world!" to the screen
```

C.3.2 #define

Constants are named using the `#define` directive and then used by name throughout the program. These globally defined constants are also called *macros*. For example, suppose you write a program that allows at most 5 user guesses, you can use `#define` to identify that number.

```
#define MAXGUESSES 5
```

Number constants in C default to decimal but can also be hexadecimal (prefix "0x") or octal (prefix "0"). Binary constants are not defined in C99 but are supported by some compilers (prefix "0b"). For example, the following assignments are equivalent:

```
char x = 37;
char x = 0x25;
char x = 045;
```

The `#` indicates that this line in the program will be handled by the *preprocessor*. Before compilation, the preprocessor replaces each occurrence of the identifier `MAXGUESSES` in the program with `5`. By convention, `#define` lines are located at the top of the file and identifiers are written in all capital letters. By defining constants in one location and then using the identifier in the program, the program remains consistent, and the value is easily modified – it need only be changed at the `#define` line instead of at each line in the code where the value is needed.

C Code Example C.2 shows how to use the `#define` directive to convert inches to centimeters. The variables `inch` and `cm` are declared to be `float` to indicate they represent single-precision floating point numbers. If the conversion factor (`INCH2CM`) were used throughout a large program, having it declared using `#define` obviates errors due to typos (for example, typing 2.53 instead of 2.54) and makes it easy to find and change (for example, if more significant digits were required).

Globally defined constants eradicate *magic numbers* from a program. A magic number is a constant that shows up in a program without a name. The presence of magic numbers in a program often introduces tricky bugs – for example, when the number is changed in one location but not another.

C Code Example C.2 USING `#define` TO DECLARE CONSTANTS

```
// Convert inches to centimeters
#include <stdio.h>

#define INCH2CM 2.54

int main(void) {
  float inch = 5.5;     // 5.5 inches
  float cm;

  cm = inch * INCH2CM;
  printf("%f inches = %f cm\n", inch, cm);
}
```

Console Output

```
5.500000 inches = 13.970000 cm
```

C.3.3 `#include`

Modularity encourages us to split programs across separate files and functions. Commonly used functions can be grouped together for easy reuse. Variable declarations, defined values, and function definitions located in a *header file* can be used by another file by adding the `#include` preprocesser directive. *Standard libraries* that provide commonly used functions are accessed in this way. For example, the following line is required to use the functions defined in the standard input/output (I/O) library, such as `printf`.

```
#include <stdio.h>
```

The "`.h`" postfix of the include file indicates it is a header file. While `#include` directives can be placed anywhere in the file before the included functions, variables, or identifiers are needed, they are conventionally located at the top of a C file.

Programmer-created header files can also be included by using quotation marks (`" "`) around the file name instead of brackets (`< >`). For example, a user-created header file called `myfunctions.h` would be included using the following line.

```
#include "myfunctions.h"
```

At compile time, files specified in brackets are searched for in system directories. Files specified in quotes are searched for in the same local directory where the C file is found. If the user-created header file is located in a different directory, the path of the file relative to the current directory must be included.

SUMMARY

- **Comments:** C provides single-line comments (`//`) and multi-line comments (`/* */`).
- **`#define NAME val`:** the `#define` directive allows an identifier (`NAME`) to be used throughout the program. Before compilation, all instances of `NAME` are replaced with `val`.
- **`#include`:** `#include` allows common functions to be used in a program. For built-in libraries, include the following line at the top of the code: `#include <library.h>` To include a user-defined header file, the name must be in quotes, listing the path relative to the current directory as needed: i.e., `#include "other/myFuncs.h"`.

C.4 VARIABLES

Variables in C programs have a type, name, value, and memory location. A variable declaration states the type and name of the variable. For example, the following declaration states that the variable is of type `char` (which is a 1-byte type), and that the variable name is `x`. The compiler decides where to place this 1-byte variable in memory.

```
char x;
```

C views memory as a group of consecutive bytes, where each byte of memory is assigned a unique number indicating its location or *address*, as shown in Figure C.1. A variable occupies one or more bytes of memory, and the address of multiple-byte variables is indicated by the lowest numbered byte. The type of a variable indicates whether to interpret the byte(s) as an integer, floating point number, or other type. The rest of this section describes C's primitive data types, the declaration of global and local variables, and the initialization of variables.

Variable names are case sensitive and can be of your choosing. However, the name may not be any of C's reserved words (i.e., `int`, `while`, etc.), may not start with a number (i.e., `int 1x;` is not a valid declaration), and may not include special characters such as `\`, `*`, `?`, or `-`. Underscores (_) are allowed.

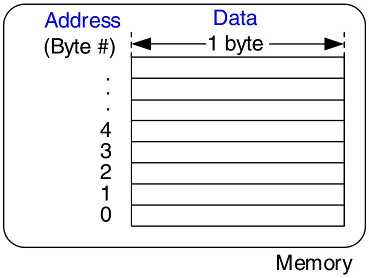


Figure C.1 C's view of memory

C.4.1 Primitive Data Types

C has a number of primitive, or built-in, data types available. They can be broadly characterized as integers, floating-point variables, and characters. An integer represents a 2's complement or unsigned number within a finite range. A floating-point variable uses IEEE floating point representation to describe real numbers with a finite range and precision. A character can be viewed as either an ASCII value or an 8-bit integer.[1] Table C.2 lists the size and range of each primitive type. Integers may be 16, 32, or 64 bits. They use 2's complement unless qualified as `unsigned`. The size

Table C.2 Primitive data types and sizes

Type	Size (bits)	Minimum	Maximum
`char`	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
`unsigned char`	8	0	$2^8 - 1 = 255$
`short`	16	$-2^{15} = -32{,}768$	$2^{15} - 1 = 32{,}767$
`unsigned short`	16	0	$2^{16} - 1 = 65{,}535$
`long`	32	$-2^{31} = -2{,}147{,}483{,}648$	$2^{31} - 1 = 2{,}147{,}483{,}647$
`unsigned long`	32	0	$2^{32} - 1 = 4{,}294{,}967{,}295$
`long long`	64	-2^{63}	$2^{63} - 1$
`unsigned long`	64	0	$2^{64} - 1$
`int`	machine-dependent		
`unsigned int`	machine-dependent		
`float`	32	$\pm 2^{-126}$	$\pm 2^{127}$
`double`	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

---

[1] Technically, the C99 standard defines a character as "a bit representation that fits in a byte," without requiring a byte to be 8 bits. However, current systems define a byte as 8 bits.

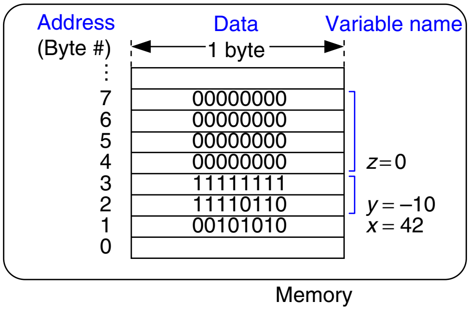


Figure C.2 **Variable storage in memory for C Code Example C.3**

of the `int` type is machine dependent and is generally the native word size of the machine. For example, on a 32-bit MIPS processor, the size of `int` or `unsigned int` is 32 bits. Floating point numbers may be 32- or 64-bit single or double precision. Characters are 8 bits.

C Code Example C.3 shows the declaration of variables of different types. As shown in Figure C.2, `x` requires one byte of data, `y` requires two, and `z` requires four. The program decides where these bytes are stored in memory, but each type always requires the same amount of data. For illustration, the addresses of `x`, `y`, and `z` in this example are 1, 2, and 4. Variable names are case-sensitive, so, for example, the variable `x` and the variable `X` are two different variables. (But it would be very confusing to use both in the same program!)

The machine-dependent nature of the `int` data type is a blessing and a curse. On the bright side, it matches the native word size of the processor so it can be fetched and manipulated efficiently. On the down side, programs using `int`s may behave differently on different computers. For example, a banking program might store the number of cents in your bank account as an `int`. When compiled on a 64-bit PC, it will have plenty of range for even the wealthiest entrepreneur. But if it is ported to a 16-bit microcontroller, it will overflow for accounts exceeding $327.67, resulting in unhappy and poverty-stricken customers.

C Code Example C.3 EXAMPLE DATA TYPES

```
// Examples of several data types and their binary representations
unsigned char x = 42;        // x = 00101010
short y = −10;               // y = 11111111 11110110
unsigned long z = 0;         // z = 00000000 00000000 00000000 00000000
```

C.4.2 Global and Local Variables

Global and local variables differ in where they are declared and where they are visible. A global variable is declared outside of all functions, typically at the top of a program, and can be accessed by all functions. Global variables should be used sparingly because they violate the principle of modularity, making large programs more difficult to read. However, a variable accessed by many functions can be made global.

A local variable is declared inside a function and can only be used by that function. Therefore, two functions can have local variables with the same names without interfering with each other. Local variables are declared at the beginning of a function. They cease to exist when the function ends and are recreated when the function is called again. They do not retain their value from one invocation of a function to the next.

The *scope* of a variable is the context in which it can be used. For example, for a local variable, its scope is the function in which it is declared. It is out of scope everywhere else.

C Code Examples C.4 and C.5 compare programs using global versus local variables. In C Code Example C.4, the global variable `max` can be accessed by any function. Using a local variable, as shown in C Code Example C.5, is the preferred style because it preserves the well-defined interface of modularity.

C Code Example C.4 GLOBAL VARIABLES

```
// Use a global variable to find and print the maximum of 3 numbers

int max;              // global variable holding the maximum value

void findMax(int a, int b, int c) {
  max = a;
  if (b > max) {
    if (c > b) max = c;
    else       max = b;
  } else if (c > max) max = c;
}

void printMax(void) {
  printf("The maximum number is: %d\n", max);
}

int main(void) {
  findMax(4, 3, 7);
  printMax();
}
```

C Code Example C.5 LOCAL VARIABLES

```
// Use local variables to find and print the maximum of 3 numbers

int getMax(int a, int b, int c) {
  int result = a;  // local variable holding the maximum value

  if (b > result) {
    if (c > b) result = c;
    else       result = b;
  } else if (c > result) result = c;

  return result;
}

void printMax(int m) {
  printf("The maximum number is: %d\n", m);
}

int main(void) {
  int max;

  max = getMax(4, 3, 7);
  printMax(max);
}
```

C.4.3 Initializing Variables

A variable needs to be *initialized* – assigned a value – before it is read. When a variable is declared, the correct number of bytes is reserved for that variable in memory. However, the memory at those locations retains whatever value it had last time it was used, essentially a random value. Global and local variables can be initialized either when they are declared or within the body of the program. C Code Example C.3 shows variables initialized at the same time they are declared. C Code Example C.4 shows how variables are initialized before their use, but after declaration; the global variable `max` is initialized by the `getMax` function before it is read by the `printMax` function. Reading from uninitialized variables is a common programming error, and can be tricky to debug.

SUMMARY

- **Variables:** Each variable is defined by its data type, name, and memory location. A variable is declared as `datatype name`.
- **Data types:** A data type describes the size (number of bytes) and representation (interpretation of the bytes) of a variable. Table C.2 lists C's built-in data types.
- **Memory:** C views memory as a list of bytes. Memory stores variables and associates each variable with an address (byte number).
- **Global variables:** Global variables are declared outside of all functions and can be accessed anywhere in the program.
- **Local variables:** Local variables are declared within a function and can be accessed only within that function.
- **Variable initialization:** Each variable must be initialized before it is read. Initialization can happen either at declaration or afterward.

C.5 OPERATORS

The most common type of statement in a C program is an *expression*, such as

```
y = a + 3;
```

An expression involves operators (such as + or *) acting on one or more operands, such as variables or constants. C supports the operators shown in Table C.3, listed by category and in order of decreasing precedence. For example, multiplicative operators take precedence over additive

Table C.3 Operators listed by decreasing precedence

Category	Operator	Description	Example
Unary	`++`	post-increment	`a++; // a = a+1`
	`--`	post-decrement	`x--; // x = x–1`

	`&`	memory address of a variable	`x = &y;   // x = the memory`
			`// address of y`

	`~`	bitwise NOT	`z = ~a;`
	`!`	Boolean NOT	`!x`
	`-`	negation	`y = -a;`
	`++`	pre-increment	`++a; // a = a+1`
	`--`	pre-decrement	`--x; // x = x–1`

	`(type)`	casts a variable to (type)	`x = (int)c; // cast c to an`
			`// int and assign it to x`

	`sizeof()`	size of a variable or type in bytes	`long int y;` `x = sizeof(y); // x = 4`
Multiplicative	`*`	multiplication	`y = x * 12;`
	`/`	division	`z = 9 / 3; // z = 3`
	`%`	modulo	`z = 5 % 2; // z = 1`
Additive	`+`	addition	`y = a + 2;`
	`-`	subtraction	`y = a - 2;`
Bitwise Shift	`<<`	bitshift left	`z = 5 << 2; // z = 0b00010100`
	`>>`	bitshift right	`x = 9 >> 3; // x = 0b00000001`
Relational	`==`	equals	`y == 2`
	`!=`	not equals	`x != 7`
	`<`	less than	`y < 12`
	`>`	greater than	`val > max`
	`<=`	less than or equal	`z <= 2`
	`>=`	greater than or equal	`y >= 10`

Table C.3 Operators listed by decreasing precedence—Cont'd

Category	Operator	Description	Example
Bitwise	&	bitwise AND	y = a & 15;
	^	bitwise XOR	y = 2 ^ 3;
	|	bitwise OR	y = a | b;
Logical	&&	Boolean AND	x && y
	||	Boolean OR	x || y
Ternary	? :	ternary operator	y = x ? a : b; // if x is TRUE, // y=a, else y=b
Assignment	=	assignment	x = 22;
	+=	addition and assignment	y += 3; // y = y + 3
	−=	subtraction and assignment	z −= 10; // z = z - 10
	*=	multiplication and assignment	x *= 4; // x = x * 4
	/=	division and assignment	y /= 10; // y = y / 10
	%=	modulo and assignment	x %= 4; // x = x % 4
	>>=	bitwise right-shift and assignment	x >>= 5; // x = x>>5
	<<=	bitwise left-shift and assignment	x <<= 2; // x = x<<2
	&=	bitwise AND and assignment	y &= 15; // y = y & 15
	|=	bitwise OR and assignment	x |= y; // x = x | y
	^=	bitwise XOR and assignment	x ^= y; // x = x ^ y

operators. Within the same category, operators are evaluated in the order that they appear in the program.

Unary operators, also called monadic operators, have a single operand. Ternary operators have three operands, and all others have two. The ternary operator (from the Latin *ternarius* meaning consisting of three) chooses the second or third operand depending on whether the first value is TRUE (nonzero) or FALSE (zero), respectively. C Code Example C.6 shows how to compute `y = max(a,b)` using the ternary operator, along with an equivalent but more verbose `if/else` statement.

The Truth, the Whole Truth, and Nothing But the Truth
C considers a variable to be TRUE if it is nonzero and FALSE if it is zero. Logical and ternary operators, as well as control-flow statements such as `if` and `while`, depend on the truth of a variable. Relational and logical operators produce a result that is 1 when TRUE or 0 when FALSE.

C Code Example C.6 (a) TERNARY OPERATOR, AND (b) EQUIVALENT `if/else` STATEMENT

```
(a) y = (a > b) ? a : b; // parentheses not necessary, but makes it clearer

(b) if (a > b) y = a;
    else       y = b;
```

Simple assignment uses the = operator. C code also allows for compound assignment, that is, assignment after a simple operation such as addition (+=) or multiplication (*=). In compound assignments, the variable on the left side is both operated on and assigned the result. C Code Example C.7 shows these and other C operations. Binary values in the comments are indicated with the prefix “`0b`”.

C Code Example C.7 OPERATOR EXAMPLES

<table>
<tr><th>Expression</th><th>Result</th><th>Notes</th></tr>
<tr><td>44 / 14</td><td>3</td><td>Integer division truncates</td></tr>
<tr><td>44 % 14</td><td>2</td><td>44 mod 14</td></tr>
<tr><td>0x2C && 0xE   //0b101100 && 0b1110</td><td>1</td><td>Logical AND</td></tr>
<tr><td>0x2C || 0xE   //0b101100 || 0b1110</td><td>1</td><td>Logical OR</td></tr>
<tr><td>0x2C & 0xE   //0b101100 & 0b1110</td><td>0xC (0b001100)</td><td>Bitwise AND</td></tr>
<tr><td>0x2C | 0xE   //0b101100 | 0b1110</td><td>0x2E (0b101110)</td><td>Bitwise OR</td></tr>
<tr><td>0x2C ^ 0xE   //0b101100 ^ 0b1110</td><td>0x22 (0b100010)</td><td>Bitwise XOR</td></tr>
<tr><td>0xE << 2   //0b1110 << 2</td><td>0x38 (0b111000)</td><td>Left shift by 2</td></tr>
<tr><td>0x2C >> 3   //0b101100 >> 3</td><td>0x5 (0b101)</td><td>Right shift by 3</td></tr>
<tr><td>x = 14;
x += 2;</td><td>x=16</td><td></td></tr>
<tr><td>y = 0x2C;   // y = 0b101100
y &= 0xF;   // y &= 0b1111</td><td>y=0xC (0b001100)</td><td></td></tr>
<tr><td>x = 14; y = 44;
y = y + x++;</td><td>x=15, y=58</td><td>Increment x after using it</td></tr>
<tr><td>x = 14; y = 44;
y = y + ++x;</td><td>x=15, y=59</td><td>Increment x before using it</td></tr>
</table>

C.6 FUNCTION CALLS

Modularity is key to good programming. A large program is divided into smaller parts called functions that, similar to hardware modules, have well-defined inputs, outputs, and behavior. C Code Example C.8 shows the `sum3` function. The function declaration begins with the return type, `int`, followed by the name, `sum3`, and the inputs enclosed within parentheses `(int a, int b, int c)`. Curly braces `{}` enclose the body of the function, which may contain zero or more statements. The `return` statement indicates the value that the function should return to its caller; this can be viewed as the output of the function. A function can only return a single value.

C Code Example C.8 `sum3` **FUNCTION**

```
// Return the sum of the three input variables
int sum3(int a, int b, int c) {
  int result = a + b + c;
  return result;
}
```

After the following call to `sum3`, `y` holds the value 42.

```
int y = sum3(10, 15, 17);
```

Although a function may have inputs and outputs, neither is required. C Code Example C.9 shows a function with no inputs or outputs. The keyword `void` before the function name indicates that nothing is returned. `void` between the parentheses indicates that the function has no input arguments.

Nothing between the parentheses also indicates no input arguments. So, in this case we could have written:

`void printPrompt()`

C Code Example C.9 **FUNCTION** `printPrompt` **WITH NO INPUTS OR OUTPUTS**

```
// Print a prompt to the console
void printPrompt(void)
{
  printf("Please enter a number from 1-3:\n");
}
```

A function must be declared in the code before it is called. This may be done by placing the called function earlier in the file. For this reason, `main` is often placed at the end of the C file after all the functions it calls. Alternatively, a function *prototype* can be placed in the program before the function is defined. The function prototype is the first line of the function, declaring the return type, function name, and function inputs. For example, the function prototypes for the functions in C Code Examples C.8 and C.9 are:

```
int sum3(int a, int b, int c);
void printPrompt(void);
```

With careful ordering of functions, prototypes may be unnecessary. However, they are unavoidable in certain cases, such as when function `f1` calls `f2` and `f2` calls `f1`. It is good style to place prototypes for all of a program's functions near the beginning of the C file or in a header file.

C Code Example C.10 shows how function prototypes are used. Even though the functions themselves are after `main`, the function prototypes at the top of the file allow them to be used in `main`.

C Code Example C.10 FUNCTION PROTOTYPES

```
#include <stdio.h>

// function prototypes
int sum3(int a, int b, int c);
void printPrompt(void);

int main(void)
{
  int y = sum3(10, 15, 20);

  printf("sum3 result: %d\n", y);
  printPrompt();
}

int sum3(int a, int b, int c) {
  int result = a+b+c;
  return result;
}

void printPrompt(void) {
  printf("Please enter a number from 1-3:\n");
}
```

Console Output

```
sum3 result: 45
Please enter a number from 1-3:
```

As with variable names, function names are case sensitive, cannot be any of C's reserved words, may not contain special characters (except underscore _), and cannot start with a number. Typically function names include a verb to indicate what they do.

Be consistent in how you capitalize your function and variable names so you don't have to constantly look up the correct capitalization. Two common styles are to camelCase, in which the initial letter of each word after the first is capitalized like the humps of a camel (e.g., `printPrompt`), or to use underscores between words (e.g., `print_prompt`). We have unscientifically observed that reaching for the underscore key exacerbates carpal tunnel syndrome (my pinky finger twinges just thinking about the underscore!) and hence prefer camelCase. But the most important thing is to be consistent in style within your organization.

The `main` function is always declared to return an `int`, which conveys to the operating system the reason for program termination. A zero indicates normal completion, while a nonzero value signals an error condition. If `main` reaches the end without encountering a `return` statement, it will automatically return 0. Most operating systems do not automatically inform the user of the value returned by the program.

C.7 CONTROL-FLOW STATEMENTS

C provides control-flow statements for conditionals and loops. Conditionals execute a statement only if a condition is met. A loop repeatedly executes a statement as long as a condition is met.

C.7.1 Conditional Statements

`if`, `if/else`, and `switch/case` statements are conditional statements commonly used in high-level languages including C.

if Statements

An `if` statement executes the statement immediately following it when the expression in parentheses is TRUE (i.e., nonzero). The general format is:

```
if (expression)
  statement
```

C Code Example C.11 shows how to use an `if` statement in C. When the variable `aintBroke` is equal to 1, the variable `dontFix` is set to 1. A block of multiple statements can be executed by placing curly braces `{}` around the statements, as shown in C Code Example C.12.

Curly braces, `{}`, are used to group one or more statements into a *compound statement* or *block*.

C Code Example C.11 `if` STATEMENT

```
int dontFix = 0;

if (aintBroke == 1)
  dontFix = 1;
```

C Code Example C.12 `if` STATEMENT WITH A BLOCK OF CODE

```
// If amt >= $2, prompt user and dispense candy
if (amt >= 2) {
  printf("Select candy.\n");
  dispenseCandy = 1;
}
```

if/else Statements

`if/else` statements execute one of two statements depending on a condition, as shown below. When the `expression` in the `if` statement is TRUE, `statement1` is executed. Otherwise, `statement2` is executed.

```
if (expression)
  statement1
else
  statement2
```

C Code Example C.6(b) gives an example `if/else` statement in C. The code sets `max` equal to `a` if `a` is greater than `b`; otherwise `max = b`.

switch/case Statements

`switch/case` statements execute one of several statements depending on the conditions, as shown in the general format below.

```
switch (variable) {
  case (expression1): statement1 break;
  case (expression2): statement2 break;
  case (expression3): statement3 break;
  default:            statement4
}
```

For example, if `variable` is equal to `expression2`, execution continues at `statement2` until the keyword `break` is reached, at which point it exits the `switch/case` statement. If no conditions are met, the `default` executes.

If the keyword `break` is omitted, execution begins at the point where the condition is TRUE and then falls through to execute the remaining cases below it. This is usually not what you want and is a common error among beginning C programmers.

C Code Example C.13 shows a `switch/case` statement that, depending on the variable `option`, determines the amount of money `amt` to be disbursed. A `switch/case` statement is equivalent to a series of nested `if/else` statements, as shown by the equivalent code in C Code Example C.14.

C Code Example C.13 `switch/case` STATEMENT

```
// Assign amt depending on the value of option
switch (option) {
  case 1:  amt = 100; break;
  case 2:  amt = 50;  break;
  case 3:  amt = 20;  break;
  case 4:  amt = 10;  break;
  default: printf("Error: unknown option.\n");
}
```

C Code Example C.14 NESTED `if/else` STATEMENT

```
// Assign amt depending on the value of option
if      (option == 1)  amt = 100;
else if (option == 2)  amt = 50;
else if (option == 3)  amt = 20;
else if (option == 4)  amt = 10;
else printf("Error: unknown option.\n");
```

C.7.2 Loops

`while`, `do/while`, and `for` loops are common loop constructs used in many high-level languages including C. These loops repeatedly execute a statement as long as a condition is satisfied.

while Loops

`while` loops repeatedly execute a statement until a condition is not met, as shown in the general format below.

```
while (condition)
  statement
```

The `while` loop in C Code Example C.15 computes the factorial of 9 = 9 × 8 × 7 × ⋯ × 1. Note that the condition is checked before executing the `statement`. In this example, the statement is a compound statement or block, so curly braces are required.

C Code Example C.15 `while` LOOP

```
// Compute 9! (the factorial of 9)
int i = 1, fact = 1;

// multiply the numbers from 1 to 9
while (i < 10) { // while loops check the condition first
  fact *= i;
  i++;
}
```

do/while Loops

`do/while` loops are like `while` loops but the `condition` is checked only after the `statement` is executed once. The general format is shown below. The `condition` is followed by a semi-colon.

```
do
  statement
while (condition);
```

The `do/while` loop in C Code Example C.16 queries a user to guess a number. The program checks the condition (if the user's number is equal to the correct number) only after the body of the `do/while` loop executes once. This construct is useful when, as in this case, something must be done (for example, the guess retrieved from the user) before the condition is checked.

C Code Example C.16 `do/while` LOOP

```
// Query user to guess a number and check it against the correct number.
#define MAXGUESSES 3
#define CORRECTNUM 7
int guess, numGuesses = 0;
```

```
do {
  printf("Guess a number between 0 and 9. You have %d more guesses.\n",
         (MAXGUESSES-numGuesses));
  scanf("%d", &guess);     // read user input
  numGuesses++;
} while ( (numGuesses < MAXGUESSES) & (guess != CORRECTNUM) );
// do loop checks the condition after the first iteration

if (guess == CORRECTNUM)
  printf("You guessed the correct number!\n");
```

`for` Loops

`for` loops, like `while` and `do/while` loops, repeatedly execute a `statement` until a condition is not satisfied. However, `for` loops add support for a *loop variable*, which typically keeps track of the number of loop executions. The general format of the `for` loop is

```
for (initialization; condition; loop operation)
  statement
```

The `initialization` code executes only once, before the `for` loop begins. The `condition` is tested at the beginning of each iteration of the loop. If the `condition` is not TRUE, the loop exits. The `loop operation` executes at the end of each iteration. C Code Example C.17 shows the factorial of 9 computed using a `for` loop.

C Code Example C.17 `for` LOOP

```
// Compute 9!
int i;    // loop variable
int fact = 1;

for (i=1; i<10; i++)
  fact *= i;
```

Whereas the `while` and `do/while` loops in C Code Examples C.15 and C.16 include code for incrementing and checking the loop variable `i` and `numGuesses`, respectively, the `for` loop incorporates those statements into its format. A `for` loop could be expressed equivalently, but less conveniently, as

```
initialization;
while (condition) {
  statement
  loop operation;
}
```

SUMMARY

- **Control-flow statements:** C provides control-flow statements for conditional statements and loops.
- **Conditional statements:** Conditional statements execute a statement when a condition is TRUE. C includes the following conditional statements: `if`, `if/else`, and `switch/case`.
- **Loops:** Loops repeatedly execute a statement until a condition is FALSE. C provides `while`, `do/while`, and `for` loops.

C.8 MORE DATA TYPES

Beyond various sizes of integers and floating-point numbers, C includes other special data types including pointers, arrays, strings, and structures. These data types are introduced in this section along with dynamic memory allocation.

C.8.1 Pointers

A pointer is the address of a variable. C Code Example C.18 shows how to use pointers. `salary1` and `salary2` are variables that can contain integers, and `ptr` is a variable that can hold the address of an integer. The compiler will assign arbitrary locations in RAM for these variables depending on the runtime environment. For the sake of concreteness, suppose this program is compiled on a 32-bit system with `salary1` at addresses 0x70-73, `salary2` at addresses 0x74-77, and `ptr` at 0x78-7B. Figure C.3 shows memory and its contents after the program is executed.

In a variable declaration, a star (`*`) before a variable name indicates that the variable is a pointer to the declared type. In using a pointer variable, the `*` operator *dereferences* a pointer, returning the value stored at the

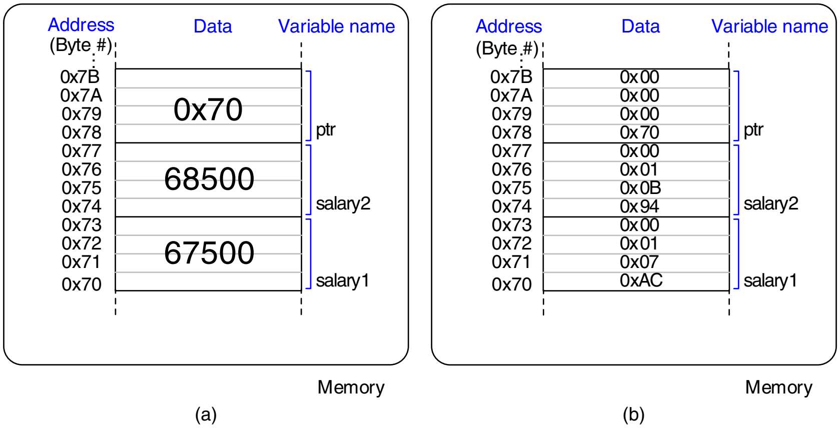

Address
(Byte #)
Data
Variable name
0x7B
0x7A
0x79
0x78
0x77
0x76
0x75
0x74
0x73
0x72
0x71
0x70
0x70
68500
67500
ptr
salary2
salary1
Memory
(a)
Address
(Byte #)
Data
Variable name
0x7B 0x00
0x7A 0x00
0x79 0x00
0x78 0x70
0x77 0x00
0x76 0x01
0x75 0x0B
0x74 0x94
0x73 0x00
0x72 0x01
0x71 0x07
0x70 0xAC
ptr
salary2
salary1
Memory
(b)

Figure C.3 Contents of memory after C Code Example C.18 executes shown (a) by value and (b) by byte using little-endian memory

Dereferencing a pointer to a non-existent memory location or an address outside of the range accessible by the program will usually cause a program to crash. The crash is often called a *segmentation fault*.

indicated memory address contained in the pointer. The `&` operator is pronounced "address of," and it produces the memory address of the variable being referenced.

Pointers are particularly useful when a function needs to modify a variable, instead of just returning a value. Because functions can't modify their inputs directly, a function can make the input a pointer to the variable. This is called passing an input variable *by reference* instead of *by value*, as shown in prior examples. C Code Example C.19 gives an example of passing `x` by reference so that `quadruple` can modify the variable directly.

C Code Example C.18 POINTERS

```
// Example pointer manipulations
int salary1, salary2;  // 32-bit numbers
int *ptr;              // a pointer specifying the address of an int variable

salary1 = 67500;       // salary1 = $67,500 = 0x000107AC
ptr = &salary1;        // ptr = 0x0070, the address of salary1
salary2 = *ptr + 1000; /* dereference ptr to give the contents of address 70 = $67,500,
                          then add $1,000 and set salary2 to $68,500 */
```

C Code Example C.19 PASSING AN INPUT VARIABLE BY REFERENCE

```
// Quadruple the value pointed to by a
#include <stdio.h>

void quadruple(int *a)
{
  *a = *a * 4;
}

int main(void)
{
  int x = 5;

  printf("x before: %d\n", x);
  quadruple(&x);
  printf("x after: %d\n", x);
  return 0;
}
```

Console Output

```
x before: 5
x after: 20
```

A pointer to address 0 is called a *null pointer* and indicates that the pointer is not actually pointing to meaningful data. It is written as NULL in a program.

C.8.2 Arrays

An array is a group of similar variables stored in consecutive addresses in memory. The elements are numbered from 0 to $N-1$, where N is the size of the array. C Code Example C.20 declares an array variable called `scores` that holds the final exam scores for three students. Memory space is reserved for three `long`s, that is, 3 × 4 = 12 bytes. Suppose the `scores` array starts at address 0x40. The address of the 1st element (i.e., `scores[0]`) is 0x40, the 2nd element is 0x44, and the 3rd element is 0x48, as shown in Figure C.4. In C, the array variable, in this case `scores`, is a pointer to the 1st element. It is the programmer's responsibility not to access elements beyond the end of the array. C has no internal bounds checking, so a program that writes beyond the end of an array will compile fine but may stomp on other parts of memory when it runs.

C Code Example C.20 ARRAY DECLARATION

```
long scores[3];    // array of three 4-byte numbers
```

The elements of an array can be initialized either at declaration using curly braces `{}`, as shown in C Code Example C.21, or individually in the body of the code, as shown in C Code Example C.22. Each element of an array is accessed using brackets `[]`. The contents of memory containing the array are shown in Figure C.4. Array initialization using curly braces `{}` can only be performed at declaration, and not afterward. `for` loops are commonly used to assign and read array data, as shown in C Code Example C.23.

C Code Example C.21 ARRAY INITIALIZATION AT DECLARATION USING { }

```
long scores[3]={93, 81, 97}; // scores[0]=93; scores[1]=81; scores[2]=97;
```

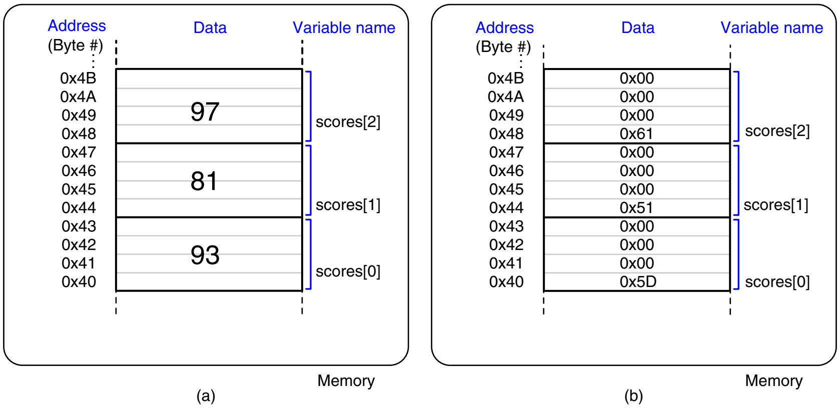

Address
(Byte #)
Data
Variable name
0x4B
0x4A
0x49
0x48
0x47
0x46
0x45
0x44
0x43
0x42
0x41
0x40
97
81
93
scores[2]
scores[1]
scores[0]
Memory
(a)
Address
(Byte #)
Data
Variable name
0x4B 0x00
0x4A 0x00
0x49 0x00
0x48 0x61
0x47 0x00
0x46 0x00
0x45 0x00
0x44 0x51
0x43 0x00
0x42 0x00
0x41 0x00
0x40 0x5D
scores[2]
scores[1]
scores[0]
Memory
(b)

Figure C.4 `scores` **array stored in memory**

C Code Example C.22 ARRAY INITIALIZATION USING ASSIGNMENT

```
long scores[3];

scores[0] = 93;
scores[1] = 81;
scores[2] = 97;
```

C Code Example C.23 ARRAY INITIALIZATION USING A `for` LOOP

```
// User enters 3 student scores into an array
long scores[3];
int i, entered;

printf("Please enter the student's 3 scores.\n");
for (i=0; i<3; i++) {
  printf("Enter a score and press enter.\n");
  scanf("%d", &entered);
  scores[i] = entered;
}
printf("Scores: %d %d %d\n", scores[0], scores[1], scores[2]);
```

When an array is declared, the length must be constant so that the compiler can allocate the proper amount of memory. However, when the array is passed to a function as an input argument, the length need not be defined because the function only needs to know the address of the beginning of the array. C Code Example C.24 shows how an array is passed to a function. The input argument `arr` is simply the address of the 1st element of an array. Often the number of elements in an array is also passed as an input argument. In a function, an input argument of type `int[]` indicates that it is an array of integers. Arrays of any type may be passed to a function.

C Code Example C.24 PASSING AN ARRAY AS AN INPUT ARGUMENT

```
// Initialize a 5-element array, compute the mean, and print the result.
#include <stdio.h>

// Returns the mean value of an array (arr) of length len
float getMean(int arr[], int len) {
  int i;
  float mean, total = 0;

  for (i=0; i < len; i++)
    total += arr[i];

  mean = total / len;
  return mean;
}
```

```
int main(void) {
  int data[4] = {78, 14, 99, 27};
  float avg;

  avg = getMean(data, 4);

  printf("The average value is: %f.\n", avg);
}
```

Console Output

```
The average value is: 54.500000.
```

An array argument is equivalent to a pointer to the beginning of the array. Thus, `getMean` could also have been declared as

```
float getMean(int *arr, int len);
```

Although functionally equivalent, `datatype[]` is the preferred method for passing arrays as input arguments because it more clearly indicates that the argument is an array.

A function is limited to a single output, i.e., return variable. However, by receiving an array as an input argument, a function can essentially output more than a single value by changing the array itself. C Code Example C.25 sorts an array from lowest to highest and leaves the result in the same array. The three function prototypes below are equivalent. The length of an array in a function declaration is ignored.

```
void sort(int *vals, int len);
void sort(int vals[], int len);
void sort(int vals[100], int len);
```

C Code Example C.25 PASSING AN ARRAY AND ITS SIZE AS INPUTS

```
// Sort the elements of the array vals of length len from lowest to highest
void sort(int vals[], int len)
{
  int i, j, temp;

  for (i=0; i<len; i++) {
    for (j=i+1; j<len; j++) {
      if (vals[i] > vals[j]) {
        temp = vals[i];
        vals[i] = vals[j];
        vals[j] = temp;
      }
    }
  }
}
```

Arrays may have multiple dimensions. C Code Example C.26 uses a two-dimensional array to store the grades across eight problem sets for ten students. Recall that initialization of array values using `{}` is only allowed at declaration.

C Code Example C.26 TWO-DIMENSIONAL ARRAY INITIALIZATION

```
// Initialize 2-D array at declaration
 int grades[10][8] = { {100, 107, 99,  101, 100, 104, 109, 117},
                      {103, 101, 94,  101, 102, 106, 105, 110},
                      {101, 102, 92,  101, 100, 107, 109, 110},
                      {114, 106, 95,  101, 100, 102, 102, 100},
                      {98,  105, 97,  101, 103, 104, 109, 109},
                      {105, 103, 99,  101, 105, 104, 101, 105},
                      {103, 101, 100, 101, 108, 105, 109, 100},
                      {100, 102, 102, 101, 102, 101, 105, 102},
                      {102, 106, 110, 101, 100, 102, 120, 103},
                      {99,  107, 98,  101, 109, 104, 110, 108} };
```

C Code Example C.27 shows some functions that operate on the 2-D `grades` array from C Code Example C.26. Multi-dimensional arrays used as input arguments to a function must define all but the first dimension. Thus, the following two function prototypes are acceptable:

```
void print2dArray(int arr[10][8]);
void print2dArray(int arr[][8]);
```

C Code Example C.27 OPERATING ON MULTI-DIMENSIONAL ARRAYS

```
#include <stdio.h>

// Print the contents of a 10×8 array
void print2dArray(int arr[10][8])
{
  int i, j;

  for (i=0; i<10; i++) {          // for each of the 10 students
    printf("Row %d\n", i);
    for (j=0; j<8; j++) {
      printf("%d ", arr[i][j]); // print scores for all 8 problem sets
    }
    printf("\n");
  }
}

// Calculate the mean score of a 10×8 array
float getMean(int arr[10][8])
{
  int i, j;
  float mean, total = 0;

  // get the mean value across a 2D array
  for (i=0; i<10; i++) {
    for (j=0; j<8; j++) {
      total += arr[i][j];       // sum array values
    }
  }
  mean = total/(10*8);
  printf("Mean is: %f\n", mean);

  return mean;
}
```

Note that because an array is represented by a pointer to the initial element, C cannot copy or compare arrays using the = or == operators. Instead, you must use a loop to copy or compare each element one at a time.

C.8.3 Characters

A character (`char`) is an 8-bit variable. It can be viewed either as a 2's complement number between −128 and 127 or as an ASCII code for a letter, digit, or symbol. ASCII characters can be specified as a numeric value (in decimal, hexadecimal, etc.) or as a printable character enclosed in single quotes. For example, the letter A has the ASCII code 0x41, B=0x42, etc. Thus `'A' + 3` is 0x44, or `'D'`. Table 6.2 on page 323 lists the ASCII character encodings, and Table C.4 lists characters used to indicate formatting or special characters. Formatting codes include carriage return (`\r`), newline (`\n`), horizontal tab (`\t`), and the end of a string (`\0`). `\r` is shown for completeness but is rarely used in C programs. `\r` returns the carriage (location of typing) to the beginning (left) of the line, but any text that was there is overwritten. `\n`, instead, moves the location of typing to the beginning of a new line.[2] The *NULL* character (`'\0'`) indicates the end of a text string and is discussed next in Section C.8.4.

Table C.4 Special characters

Special Character	Hexadecimal Encoding	Description
`\r`	0x0D	carriage return
`\n`	0x0A	new line
`\t`	0x09	tab
`\0`	0x00	terminates a string
`\\`	0x5C	backslash
`\"`	0x22	double quote
`\'`	0x27	single quote
`\a`	0x07	bell

The term "carriage return" originates from typewriters that required the carriage, the contraption that holds the paper, to move to the right in order to allow typing to begin at the left side of the page. A carriage return lever, shown on the left in the figure below, is pressed so that the carriage would both move to the right and advance the paper by one line, called a line feed.

A Remington electric typewriter used by Winston Churchill. (*http://cwr.iwm.org.uk/server/show/conMediaFile.71979*)

---

[2] Windows text files use `\r\n` to represent end-of-line while UNIX-based systems use `\n`, which can cause nasty bugs when moving text files between systems.

C strings are called *null terminated* or *zero terminated* because the length is determined by looking for a zero at the end. In contrast, languages such as Pascal use the first byte to specify the string length, up to a maximum of 255 characters. This byte is called the *prefix byte* and such strings are called *P-strings*. An advantage of null-terminated strings is that the length can be arbitrarily great. An advantage of P-strings is that the length can be determined immediately without having to inspect all of the characters of the string.

C.8.4 Strings

A string is an array of characters used to store a piece of text of bounded but variable length. Each character is a byte representing the ASCII code for that letter, number, or symbol. The size of the array determines the maximum length of the string, but the actual length of the string could be shorter. In C, the length of the string is determined by looking for the null terminator (ASCII value 0x00) at the end of the string.

C Code Example C.28 shows the declaration of a 10-element character array called `greeting` that holds the string `"Hello!"`. For concreteness, suppose `greeting` starts at memory address 0x50. Figure C.5 shows the contents of memory from 0x50 to 0x59 holding the string `"Hello!"` Note that the string only uses the first seven elements of the array, even though ten elements are allocated in memory.

C Code Example C.28 STRING DECLARATION

```
char greeting[10] = "Hello!";
```

C Code Example C.29 shows an alternate declaration of the string `greeting`. The pointer `greeting`, holds the address of the 1st element of a 7-element array comprised of each of the characters in "Hello!" followed by the null terminator. The code also demonstrates how to print strings by using the `%s` format code.

C Code Example C.29 ALTERNATE STRING DECLARATION

```
char *greeting = "Hello!";
printf("greeting: %s", greeting);
```

Console Output

```
greeting: Hello!
```

Unlike primitive variables, a string cannot be set equal to another string using the equals operator, =. Each element of the character array must be individually copied from the source string to the target string. This is true for any array. C Code Example C.30 copies one string, `src`, to another, `dst`. The sizes of the arrays are not needed, because the end of the `src` string is indicated by the null terminator. However, `dst` must be large enough so that you don't stomp on other data. `strcpy` and other string manipulation functions are available in C's built-in libraries (see Section C.9.4).

C Code Example C.30 COPYING STRINGS

```
// Copy the source string, src, to the destination string, dst
void strcpy(char *dst, char *src)
{
  int i = 0;

  do {
    dst[i] = src[i];       // copy characters one byte at a time
  } while (src[i++]);      // until the null terminator is found
}
```

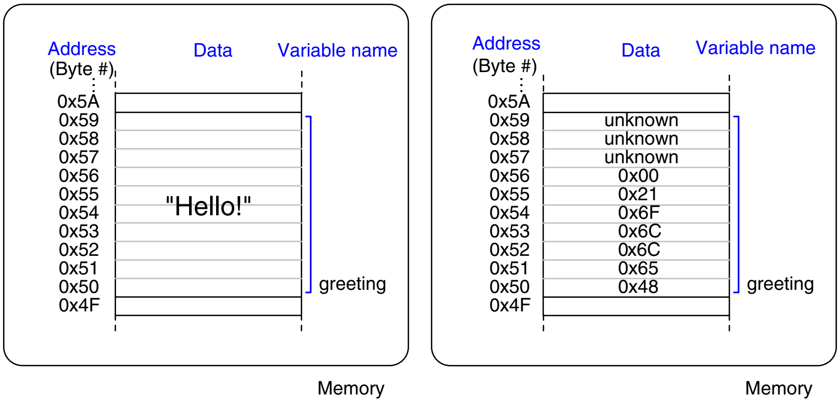

Address
(Byte #)
Data
Variable name
0x5A
0x59
0x58
0x57
0x56
0x55
0x54
0x53
0x52
0x51
0x50
0x4F
"Hello!"
greeting
Memory
Address
(Byte #)
Data
Variable name
0x5A
0x59 unknown
0x58 unknown
0x57 unknown
0x56 0x00
0x55 0x21
0x54 0x6F
0x53 0x6C
0x52 0x6C
0x51 0x65
0x50 0x48
0x4F
greeting
Memory

Figure C.5 **The string "Hello!" stored in memory.**

C.8.5 Structures

In C, structures are used to store a collection of data of various types. The general format of a structure declaration is

```
struct name {
  type1 element1;
  type2 element2;
  …
};
```

where `struct` is a keyword indicating that it is a structure, `name` is the structure tag name, and `element1` and `element2` are members of the structure. A structure may have any number of members. C Code Example C.31 shows how to use a structure to store contact information. The program then declares a variable `c1` of type `struct contact`.

C Code Example C.31 STRUCTURE DECLARATION

```
struct contact {
  char name[30];
  int phone;
  float height; // in meters
};

struct contact c1;
```

```
strcpy(c1.name, "Ben Bitdiddle");
c1.phone = 7226993;
c1.height = 1.82;
```

Just like built-in C types, you can create arrays of structures and pointers to structures. C Code Example C.32 creates an array of contacts.

C Code Example C.32 ARRAY OF STRUCTURES

```
struct contact classlist[200];
classlist[0].phone = 9642025;
```

It is common to use pointers to structures. C provides the *member access operator* -> to dereference a pointer to a structure and access a member of the structure. C Code Example C.33 shows an example of declaring a pointer to a `struct contact`, assigning it to point to the 42nd element of `classlist` from C Code Example C.32, and using the member access operator to set a value in that element.

C Code Example C.33 ACCESSING STRUCTURE MEMBERS USING POINTERS AND ->

```
struct contact *cptr;
cptr = &classlist[42];
cptr->height = 1.9;  // equivalent to: (*cptr).height = 1.9;
```

Structures can be passed as function inputs or outputs by value or by reference. Passing by value requires the compiler to copy the entire structure into memory for the function to access. This can require a large amount of memory and time for a big structure. Passing by reference involves passing a pointer to the structure, which is more efficient. The function can also modify the structure being pointed to rather than having to return another structure. C Code Example C.34 shows two versions of the `stretch` function that makes a contact 2 cm taller. `stretchByReference` avoids copying the large structure twice.

C Code Example C.34 PASSING STRUCTURES BY VALUE OR BY NAME

```
struct contact stretchByValue(struct contact c)
{
  c.height += 0.02;
  return c;
}
```

```
void stretchByReference(struct contact *cptr)
{
  cptr->height += 0.02;
}

int main(void)
{
  struct contact George;

  George.height = 1.4; // poor fellow has been stooped over
  George = stretchByValue(George); // stretch for the stars
  stretchByReference(&George);    // and stretch some more
}
```

C.8.6 * `typedef`

C also allows you to define your own names for data types using the `typedef` statement. For example, writing `struct contact` becomes tedious when it is often used, so we can define a new type named `contact` and use it as shown in C Code Example C.35.

C Code Example C.35 CREATING A CUSTOM TYPE USING `typedef`

```
typedef struct contact {
  char name[30];
  int phone;
  float height; // in meters
} contact;        // defines contact as shorthand for "struct contact"

contact c1;       // now we can declare the variable as type contact
```

`typedef` can be used to create a new type occupying the same amount of memory as a primitive type. C Code Example C.36 defines `byte` and `bool` as 8-bit types. The `byte` type may make it clearer that the purpose of `pos` is to be an 8-bit number rather than an ASCII character. The `bool` type indicates that the 8-bit number is representing TRUE or FALSE. These types make a program easier to read than if one simply used `char` everywhere.

C Code Example C.36 `typedef byte` AND `bool`

```
typedef unsigned char byte;
typedef char bool;
#define TRUE 1
#define FALSE 0

byte pos = 0x45;
bool loveC = TRUE;
```

C Code Example C.37 illustrates defining a 3-element `vector` and a 3 × 3 `matrix` type using arrays.

C Code Example C.37 `typedef vector` AND `matrix`

```
typedef double vector[3];
typedef double matrix[3][3];

vector a = {4.5, 2.3, 7.0};
matrix b = {{3.3, 4.7, 9.2}, {2.5, 4, 9}, {3.1, 99.2, 88}};
```

C.8.7 * Dynamic Memory Allocation

In all the examples thus far, variables have been declared *statically*; that is, their size is known at compile time. This can be problematic for arrays and strings of variable size because the array must be declared large enough to accommodate the largest size the program will ever see. An alternative is to *dynamically* allocate memory at run time when the actual size is known.

The `malloc` function from `stdlib.h` allocates a block of memory of a specified size and returns a pointer to it. If not enough memory is available, it returns a NULL pointer instead. For example, the following code allocates 10 shorts (10 × 2 = 20 bytes). The `sizeof` operator returns the size of a type or variable in bytes.

```
// dynamically allocate 20 bytes of memory
short *data = malloc(10*sizeof(short));
```

C Code Example C.38 illustrates dynamic allocation and de-allocation. The program accepts a variable number of inputs, stores them in a dynamically allocated array, and computes their average. The amount of memory necessary depends on the number of elements in the array and the size of each element. For example, if an `int` is a 4-byte variable and 10 elements are needed, 40 bytes are dynamically allocated. The `free` function de-allocates the memory so that it could later be used for other purposes. Failing to de-allocate dynamically allocated data is called a *memory leak* and should be avoided.

C Code Example C.38 DYNAMIC MEMORY ALLOCATION AND DE-ALLOCATION

```
// Dynamically allocate and de-allocate an array using malloc and free
#include <stdlib.h>

// Insert getMean function from C Code Example C.24.

int main(void) {
  int len, i;
  int *nums;

  printf("How many numbers would you like to enter?    ");
  scanf("%d", &len);
  nums = malloc(len*sizeof(int));
  if (nums == NULL) printf("ERROR: out of memory.\n");
  else {
    for (i=0; i<len; i++) {
      printf("Enter number:    ");
      scanf("%d", &nums[i]);
    }
    printf("The average is %f\n", getMean(nums, len));
  }
  free(nums);
}
```

C.8.8 * Linked Lists

A *linked list* is a common data structure used to store a variable number of elements. Each element in the list is a structure containing one or more data fields and a link to the next element. The first element in the list is called the *head*. Linked lists illustrate many of the concepts of structures, pointers, and dynamic memory allocation.

C Code Example C.39 describes a linked list for storing computer user accounts to accommodate a variable number of users. Each user has a user name, a password, a unique user identification number (UID), and a field indicating whether they have administrator privileges. Each element of the list is of type `userL`, containing all of this user information along with a link to the next element in the list. A pointer to the head of the list is stored in a global variable called `users`, and is initially set to NULL to indicate that there are no users.

The program defines functions to insert, delete, and find a user and to count the number of users. The `insertUser` function allocates space for a new list element and adds it to the head of the list. The `deleteUser` function scans through the list until the specified UID is found and then removes that element, adjusting the link from the previous element to skip the deleted element and freeing the memory occupied by the deleted element. The `findUser` function scans through the list until the specified UID is found and returns a pointer to that element, or NULL if the UID is not found. The `numUsers` function counts the number of elements in the list.

C Code Example C.39 LINKED LIST

```
#include <stdlib.h>
#include <string.h>

typedef struct userL {
  char uname[80];     // user name
  char passwd[80];    // password
  int uid;            // user identification number
  int admin;          // 1 indicates administrator privileges
  struct userL *next;
} userL;

userL *users = NULL;

void insertUser(char *uname, char *passwd, int uid, int admin) {
  userL *newUser;

  newUser = malloc(sizeof(userL));   // create space for new user
  strcpy(newUser->uname, uname);     // copy values into user fields
  strcpy(newUser->passwd, passwd);
  newUser->uid = uid;
  newUser->admin = admin;
  newUser->next = users;              // insert at start of linked list
  users = newUser;
}

void deleteUser(int uid) {   // delete first user with given uid
  userL *cur = users;
  userL *prev = NULL;

  while (cur != NULL) {
    if (cur->uid == uid) { // found the user to delete
      if (prev == NULL) users = cur->next;
      else prev->next = cur->next;
      free(cur);
      return; // done
    }
    prev = cur;        // otherwise, keep scanning through list
    cur = cur->next;
  }
}

userL *findUser(int uid) {
  userL *cur = users;

  while (cur != NULL) {
    if (cur->uid == uid) return cur;
    else cur = cur->next;
  }
  return NULL;
}

int numUsers(void) {
  userL *cur = users;
  int count = 0;

  while (cur != NULL) {
    count++;
    cur = cur->next;
  }
  return count;
}
```

SUMMARY

- **Pointers:** A pointer holds the address of a variable.
- **Arrays:** An array is a list of similar elements declared using square brackets `[]`.
- **Characters:** `char` types can hold small integers or special codes for representing text or symbols.
- **Strings:** A string is an array of characters ending with the null terminator 0x00.
- **Structures:** A structure stores a collection of related variables.
- **Dynamic memory allocation:** `malloc` is a built-in functions for allocating memory as the program runs. `free` de-allocates the memory after use.
- **Linked Lists:** A linked list is a common data structure for storing a variable number of elements.

C.9 STANDARD LIBRARIES

Programmers commonly use a variety of standard functions, such as printing and trigonometric operations. To save each programmer from having to write these functions from scratch, C provides *libraries* of frequently used functions. Each library has a header file and an associated object file, which is a partially compiled C file. The header file holds variable declarations, defined types, and function prototypes. The object file contains the functions themselves and is linked at compile-time to create the executable. Because the library function calls are already compiled into an object file, compile time is reduced. Table C.5 lists some of the most frequently used C libraries, and each is described briefly below.

C.9.1 stdio

The standard input/output library `stdio.h` contains commands for printing to a console, reading keyboard input, and reading and writing files. To use these functions, the library must be included at the top of the C file:

```
#include <stdio.h>
```

`printf`

The *print formatted* statement `printf` displays text to the console. Its required input argument is a string enclosed in quotes `""`. The string contains text and optional commands to print variables. Variables to be printed are listed after the string, and are printed using format codes shown in Table C.6. C Code Example C.40 gives a simple example of `printf`.

Table C.5 **Frequently used C libraries**

C Library Header File	Description
`stdio.h`	**Standard input/output library**. Includes functions for printing or reading to/from the screen or a file (`printf`, `fprintf` and `scanf`, `fscanf`) and to open and close files (`fopen` and `fclose`).
`stdlib.h`	**Standard library**. Includes functions for random number generation (`rand` and `srand`), for dynamically allocating or freeing memory (`malloc` and `free`), terminating the program early (`exit`), and for conversion between strings and numbers (`atoi`, `atol`, and `atof`).
`math.h`	**Math library**. Includes standard math functions such as `sin`, `cos`, `asin`, `acos`, `sqrt`, `log`, `log10`, `exp`, `floor`, and `ceil`.
`string.h`	**String library**. Includes functions to compare, copy, concatenate, and determine the length of strings.

Table C.6 `printf` **format codes for printing variables**

Code	Format
`%d`	Decimal
`%u`	Unsigned decimal
`%x`	Hexadecimal
`%o`	Octal
`%f`	Floating point number (`float` or `double`)
`%e`	Floating point number (`float` or `double`) in scientific notation (e.g., 1.56e7)
`%c`	Character (`char`)
`%s`	String (null-terminated array of characters)

C Code Example C.40 PRINTING TO THE CONSOLE USING `printf`

```
// Simple print function
#include <stdio.h>

int num = 42;
int main(void) {
  printf("The answer is %d.\n", num);
}
```

Console Output:

```
The answer is 42.
```

Floating point formats (`floats` and `doubles`) default to printing six digits after the decimal point. To change the precision, replace `%f` with `%w.df`, where `w` is the minimum width of the number, and `d` is the number of decimal places to print. Note that the decimal point is included in the width count. In C Code Example C.41, `pi` is printed with a total of four characters, two of which are after the decimal point: 3.14. `e` is printed with a total of eight characters, three of which are after the decimal point. Because it only has one digit before the decimal point, it is padded with three leading spaces to reach the requested width. `c` should be printed with five characters, three of which are after the decimal point. But it is too wide to fit, so the requested width is overridden while retaining the three digits after the decimal point.

C Code Example C.41 FLOATING POINT NUMBER FORMATS FOR PRINTING

```
// Print floating point numbers with different formats
float pi = 3.14159, e = 2.7182, c = 2.998e8;
printf("pi = %4.2f\ne = %8.3f\nc = %5.3f\n", pi, e, c);
```

Console Output:

```
pi = 3.14
e  =    2.718
c  = 299800000.000
```

Because % and \ are used in print formatting, to print these characters themselves, you must use the special character sequences shown in C Code Example C.42.

C Code Example C.42 PRINTING % AND \ USING `printf`

```
// How to print % and \ to the console
printf("Here are some special characters: %% \\ \n");
```

Console Output:

```
Here are some special characters: % \
```

`scanf`

The `scanf` function reads text typed on the keyboard. It uses format codes in the same way as `printf`. C Code Example C.43 shows how to use `scanf`. When the `scanf` function is encountered, the program waits until the user types a value before continuing execution. The arguments to `scanf` are a string indicating one or more format codes and pointers to the variables where the results should be stored.

C Code Example C.43 READING USER INPUT FROM THE KEYBOARD WITH `scanf`

```
// Read variables from the command line
#include <stdio.h>

int main(void)
{
  int a;
  char str[80];
  float f;

  printf("Enter an integer.\n");
  scanf("%d", &a);
  printf("Enter a floating point number.\n");
  scanf("%f", &f);
  printf("Enter a string.\n");
  scanf("%s", str);   // note no & needed: str is a pointer
}
```

File Manipulation

Many programs need to read and write files, either to manipulate data already stored in a file or to log large amounts of information. In C, the file must first be opened with the `fopen` function. It can then be read or written with `fscanf` or `fprintf` in a way analogous to reading and writing to the console. Finally, it should be closed with the `fclose` command.

The `fopen` function takes as arguments the file name and a *print mode*. It returns a *file pointer* of type `FILE*`. If `fopen` is unable to open the file, it returns NULL. This might happen when one tries to read a nonexistent file or write a file that is already opened by another program. The modes are:

`"w"`: Write to a file. If the file exists, it is overwritten.

`"r"`: Read from a file.

`"a"`: Append to the end of an existing file. If the file doesn't exist, it is created.

C Code Example C.44 shows how to open, print to, and close a file. It is good practice to always check if the file was opened successfully and to provide an error message if it was not. The `exit` function will be discussed in Section C.9.2.2. The `fprintf` function is like `printf` but it also takes the file pointer as an input argument to know which file to write. `fclose` closes the file, ensuring that all of the information is actually written to disk and freeing up file system resources.

C Code Example C.44 PRINTING TO A FILE USING `fprintf`

```
// Write "Testing file write." to result.txt
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  FILE *fptr;

  if ((fptr = fopen("result.txt", "w")) == NULL) {
    printf("Unable to open result.txt for writing.\n");
    exit(1);  // exit the program indicating unsuccessful execution
  }
  fprintf(fptr, "Testing file write.\n");
  fclose(fptr);
}
```

It is idiomatic to open a file and check if the file pointer is NULL in a single line of code, as shown in C Code Example C.44. However, you could just as easily separate the functionality into two lines:

```
fptr = fopen("result.txt", "w");
if (fptr == NULL)
  ...
```

C Code Example C.45 illustrates reading numbers from a file named data.txt using `fscanf`. The file must first be opened for reading. The program then uses the `feof` function to check if it has reached the end of the file. As long as the program is not at the end, it reads the next number and prints it to the screen. Again, the program closes the file at the end to free up resources.

C Code Example C.45 READING INPUT FROM A FILE USING `fscanf`

```
#include <stdio.h>

int main(void)
{
  FILE *fptr;
  int data;

  // read in data from input file
  if ((fptr = fopen("data.txt", "r")) == NULL) {
    printf("Unable to read data.txt\n");
    exit(1);
  }
  while (!feof(fptr)) {   // check that the end of the file hasn't been reached
    fscanf(fptr, "%d", &data);
    printf("Read data: %d\n", data);
  }
  fclose(fptr);
}
```

data.txt

```
25 32 14 89
```

Console Output:

```
Read data: 25
Read data: 32
Read data: 14
Read data: 89
```

Other Handy stdio Functions

The `sprintf` function prints characters into a string, and `sscanf` reads variables from a string. The `fgetc` function reads a single character from a file, while `fgets` reads a complete line into a string.

`fscanf` is rather limited in its ability to read and parse complex files, so it is often easier to `fgets` one line at a time and then digest that line using `sscanf` or with a loop that inspects characters one at a time using `fgetc`.

C.9.2 stdlib

The standard library `stdlib.h` provides general purpose functions including random number generation (`rand` and `srand`), dynamic memory allocation (`malloc` and `free`, already discussed in Section C.8.8), exiting the program early (`exit`), and number format conversions. To use these functions, add the following line at the top of the C file.

```
#include <stdlib.h>
```

`rand` and `srand`

`rand` returns a pseudo-random integer. Pseudo-random numbers have the statistics of random numbers but follow a deterministic pattern starting with an initial value called the *seed*. To convert the number to a particular range, use the modulo operator (`%`) as shown in C Code Example C.46 for a range of 0 to 9. The values `x` and `y` will be random but they will be the same each time this program runs. Sample console output is given below the code.

C Code Example C.46 RANDOM NUMBER GENERATION USING `rand`

```
#include <stdlib.h>
int x, y;

x = rand();        // x = a random integer
y = rand() % 10;   // y = a random number from 0 to 9
printf("x = %d, y = %d\n", x, y);
```

Console Output:

```
x = 1481765933, y = 3
```

A programmer creates a different sequence of random numbers each time a program runs by changing the seed. This is done by calling the `srand` function, which takes the seed as its input argument. As shown in C Code Example C.47, the seed itself must be random, so a typical C program assigns it by calling the `time` function, that returns the current time in seconds.

For historical reasons, the `time` function usually returns the current time in seconds relative to January 1, 1970 00:00 UTC. UTC stands for Coordinated Universal Time, which is the same as Greenwich Mean Time (GMT). This date is just after the UNIX operating system was created by a group at Bell Labs, including Dennis Ritchie and Brian Kernighan, in 1969. Similar to New Year's Eve parties, some UNIX enthusiasts hold parties to celebrate significant values returned by `time`. For example, on February 1, 2009 at 23:31:30 UTC, `time` returned 1,234,567,890. In the year 2038, 32-bit UNIX clocks will overflow into the year 1901.

C Code Example C.47 SEEDING THE RANDOM NUMBER GENERATOR USING `srand`

```
// Produce a different random number each run
#include <stdlib.h>
#include <time.h>    // needed to call time()

int main(void)
{
  int x;

  srand(time(NULL));    // seed the random number generator
  x = rand() % 10;      // random number from 0 to 9
  printf("x = %d\n", x);
}
```

`exit`

The `exit` function terminates a program early. It takes a single argument that is returned to the operating system to indicate the reason for termination. 0 indicates normal completion, while nonzero conveys an error condition.

Format Conversion: `atoi`, `atol`, `atof`

The standard library provides functions for converting ASCII strings to integers, long integers, or doubles using `atoi`, `atol`, and `atof`, respectively, as shown in C Code Example C.48. This is particularly useful when reading in mixed data (a mix of strings and numbers) from a file or when processing numeric command line arguments, as described in Section C.10.3.

C Code Example C.48 FORMAT CONVERSION

```
// Convert ASCII strings to ints, longs, and floats
#include <stdlib.h>

int main(void)
{
  int x;
  long int y;
  double z;

  x = atoi("42");
  y = atol("833");
  z = atof("3.822");

  printf("x = %d\ty = %d\tz = %f\n", x, y, z);
}
```

Console Output:

```
x = 42   y = 833   z = 3.822000
```

C.9.3 math

The math library `math.h` provides commonly used math functions such as trigonometry functions, square root, and logs. C Code Example C.49 shows how to use some of these functions. To use math functions, place the following line in the C file:

```
#include <math.h>
```

C Code Example C.49 MATH FUNCTIONS

```
// Example math functions
#include <stdio.h>
#include <math.h>

int main(void) {
  float a, b, c, d, e, f, g, h;

  a = cos(0);           // 1, note: the input argument is in radians
  b = 2 * acos(0);      // pi (acos means arc cosine)
  c = sqrt(144);        // 12
  d = exp(2);           // e^2 = 7.389056,
  e = log(7.389056);    // 2 (natural logarithm, base e)
  f = log10(1000);      // 3 (log base 10)
  g = floor(178.567);  // 178, rounds to next lowest whole number
  h = pow(2, 10);       // computes 2 raised to the 10th power

  printf("a = %.0f, b = %f, c = %.0f, d = %.0f, e = %.2f, f = %.0f, g = %.2f, h = %.2f\n",
          a, b, c, d, e, f, g, h);
}
```

Console Output:

```
a = 1, b = 3.141593, c = 12, d = 7, e = 2.00, f = 3, g = 178.00, h = 1024.00
```

C.9.4 string

The string library `string.h` provides commonly used string manipulation functions. Key functions include:

```
// copy src into dst and return dst
char *strcpy(char *dst, char *src);

// concatenate (append) src to the end of dst and return dst
char *strcat(char *dst, char *src);

// compare two strings. Return 0 if equal, nonzero otherwise
int strcmp(char *s1, char *s2);

// return the length of str, not including the null termination
int strlen(char *str);
```

C.10 COMPILER AND COMMAND LINE OPTIONS

Although we have introduced relatively simple C programs, real-world programs can consist of tens or even thousands of C files to enable modularity, readability, and multiple programmers. This section describes how to compile a program spread across multiple C files and shows how to use compiler options and command line arguments.

C.10.1 Compiling Multiple C Source Files

Multiple C files are compiled into a single executable by listing all file names on the compile line as shown below. Remember that the group of C files still must contain only one `main` function, conventionally placed in a file named `main.c`.

```
gcc main.c file2.c file3.c
```

C.10.2 Compiler Options

Compiler options allow the programmer to specify such things as output file names and formats, optimizations, etc. Compiler options are not

Table C.7 **Compiler options**

Compiler Option	Description	Example
`-o` outfile	specifies output file name	`gcc -o hello hello.c`
`-S`	create assembly language output file (not executable)	`gcc -S hello.c` this produces `hello.s`
`-v`	verbose mode – prints the compiler results and processes as compilation completes	`gcc -v hello.c`
`-Olevel`	specify the optimization level (level is typically 0 through 3), producing faster and/or smaller code at the expense of longer compile time	`gcc -O3 hello.c`
`--version`	list the version of the compiler	`gcc –version`
`--help`	list all command line options	`gcc --help`
`-Wall`	print all warnings	`gcc -Wall hello.c`

standardized, but Table C.7 lists ones that are commonly used. Each option is typically preceded by a dash (`-`) on the command line, as shown. For example, the "`-o`" option allows the programmer to specify an output file name other than the `a.out` default. A plethora of options exist; they can be viewed by typing `gcc --help` at the command line.

C.10.3 Command Line Arguments

Like other functions, `main` can also take input variables. However, unlike other functions, these arguments are specified at the command line. As shown in C Code Example C.50, `argc` stands for *argument count*, and it denotes the number of arguments on the command line. `argv` stands for *argument vector*, and it is an array of the strings found on the command line. For example, suppose the program in C Code Example C.50 is compiled into an executable called `testargs`. When the following lines are typed at the command line, `argc` has the value 4, and the array `argv` has the values `{"./testargs", "arg1", "25", "lastarg!"}`. Note that the executable name is counted as the 1st argument. The console output after typing this command is shown below C Code Example C.50.

```
gcc -o testargs testargs.c
./testargs arg1 25 lastarg!
```

Programs that need numeric arguments may convert the string arguments to numbers using the functions in `stdlib.h`.

C Code Example C.50 COMMAND LINE ARGUMENTS

```
// Print command line arguments
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i;
  for (i=0; i<argc; i++)
    printf("argv[%d] = %s\n", i, argv[i]);
}
```

Console Output:

```
argv[0] = ./testargs
argv[1] = arg1
argv[2] = 25
argv[3] = lastarg!
```

C.11 COMMON MISTAKES

As with any programming language, you are almost certain to make errors while you write nontrivial C programs. Below are descriptions of some common mistakes made when programming in C. Some of these errors are particularly troubling because they compile but do not function as the programmer intended.

C Code Mistake C.1 MISSING & IN `scanf`

Erroneous Code

```
int a;
printf("Enter an integer:\t");
scanf("%d", a); // missing & before a
```

Corrected Code:

```
int a;
printf("Enter an integer:\t");
scanf("%d", &a);
```

C Code Mistake C.2 USING = INSTEAD OF == FOR COMPARISON

Erroneous Code

```
if (x = 1)    // always evaluates as TRUE
  printf("Found!\n");
```

Corrected Code

```
if (x == 1)
  printf("Found!\n");
```

Debugging skills are acquired with practice, but here are a few hints.

- Fix bugs starting with the first error indicated by the compiler. Later errors may be downstream effects of this error. After fixing that bug, recompile and repeat until all bugs (at least those caught by the compiler!) are fixed.
- When the compiler says a valid line of code is in error, check the code above it (i.e., for missing semicolons or braces).
- When needed, split up complicated statements into multiple lines.
- Use `printf` to output intermediate results.
- When a result doesn't match expectations, start debugging the code at the *first* place it deviates from expectations.
- Look at all compiler warnings. While some warnings can be ignored, others may alert you to more subtle code errors that will compile but not run as intended.

C Code Mistake C.3 INDEXING PAST LAST ELEMENT OF ARRAY

Erroneous Code

```
int array[10];
array[10] = 42;    // index is 0-9
```

Corrected Code

```
int array[10];
array[9] = 42;
```

C Code Mistake C.4 USING = IN `#define` STATEMENT

Erroneous Code

```
// replaces NUM with "= 4" in code
#define NUM = 4
```

Corrected Code

```
#define NUM 4
```

C Code Mistake C.5 USING AN UNINITIALIZED VARIABLE

Erroneous Code

```
int i;
if (i == 10) // i is uninitialized
  ...
```

Corrected Code

```
int i = 10;
if (i == 10)
  ...
```

C Code Mistake C.6 NOT INCLUDING PATH OF USER-CREATED HEADER FILES

Erroneous Code

```
#include   "myfile.h"
```

Corrected Code

```
#include   "othercode\myfile.h"
```

C Code Mistake C.7 USING LOGICAL OPERATORS (`!`, `||`, `&&`) INSTEAD OF BITWISE (~, |, &)

Erroneous Code

```
char x=!5;     // logical NOT: x = 0
char y=5||2;   // logical OR:  y = 1
char z=5&&2;   // logical AND: z = 1
```

Corrected Code

```
char x=~5;  // bitwise NOT: x = 0b11111010
char y=5|2;// bitwise OR:  y = 0b00000111
char z=5&2;// logical AND: z = 0b00000000
```

C Code Mistake C.8 FORGETTING `break` IN A `switch/case` STATEMENT

Erroneous Code

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1;
  case 'd': direction = 2;
  case 'l': direction = 3;
  case 'r': direction = 4;
  default: direction = 0;
}
// direction = 0
```

Corrected Code

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1; break;
  case 'd': direction = 2; break;
  case 'l': direction = 3; break;
  case 'r': direction = 4; break;
  default: direction = 0;
}
// direction = 2
```

C Code Mistake C.9 MISSING CURLY BRACES `{}`

Erroneous Code

```
if (ptr == NULL)  // missing curly braces
  printf("Unable to open file.\n");
  exit(1);      // always executes
```

Corrected Code

```
if (ptr == NULL) {
  printf("Unable to open file.\n");
  exit(1);
}
```

C Code Mistake C.10 USING A FUNCTION BEFORE IT IS DECLARED

Erroneous Code

```
int main(void)
{
  test();
}

void test(void)
{...
}
```

Corrected Code

```
void test(void)
{...
}

int main(void)
{
  test();
}
```

C Code Mistake C.11 DECLARING A LOCAL AND GLOBAL VARIABLE WITH THE SAME NAME

Erroneous Code

```
int x = 5;    // global declaration of x
int test(void)
{
  int x = 3;  // local declaration of x
  ...
}
```

Corrected Code

```
int x = 5;    // global declaration of x
int test(void)
{
  int y = 3;  // local variable is y
  ...
}
```

C Code Mistake C.12 TRYING TO INITIALIZE AN ARRAY WITH `{}` AFTER DECLARATION

Erroneous Code

```
int scores[3];
scores = {93, 81, 97}; // won't compile
```

Corrected Code

```
int scores[3] = {93, 81, 97};
```

C Code Mistake C.13 ASSIGNING ONE ARRAY TO ANOTHER USING =

Erroneous Code

```
int scores[3] = {88, 79, 93};
int scores2[3];

scores2 = scores;
```

Corrected Code

```
int scores[3] = {88, 79, 93};
int scores2[3];

for (i=0; i<3; i++)
  scores2[i] = scores[i];
```

C Code Mistake C.14 FORGETTING THE SEMI-COLON AFTER A `do/while` LOOP

Erroneous Code

```
int num;
do {
  num = getNum();
} while (num < 100)     // missing ;
```

Corrected Code

```
int num;
do {
  num = getNum();
} while (num < 100);
```

C Code Mistake C.15 USING COMMAS INSTEAD OF SEMICOLONS IN `for` LOOP

Erroneous Code

```
for (i=0, i < 200, i++)
  ...
```

Corrected Code

```
for (i=0; i < 200; i++)
  ...
```

C Code Mistake C.16 INTEGER DIVISION INSTEAD OF FLOATING POINT DIVISION

Erroneous Code

```
// integer (truncated) division occurs when
// both arguments of division are integers
float x = 9 / 4;  // x = 2.0
```

Corrected Code

```
// at least one of the arguments of
// division must be a float to
// perform floating point division
float x = 9.0 / 4;  // x = 2.25
```

C Code Mistake C.17 WRITING TO AN UNINITIALIZED POINTER

Erroneous Code	Corrected Code
`int *y = 77;`	`int x, *y = &x;` `*y = 77;`

C Code Mistake C.18 GREAT EXPECTATIONS (OR LACK THEREOF)

A common beginner error is to write an entire program (usually with little modularity) and expect it to work perfectly the first time. For non-trivial programs, writing modular code and testing the individual functions along the way is essential. Debugging becomes exponentially harder and more time-consuming with complexity.

Another common error is lacking expectations. When this happens, the programmer can only verify that the code produces a result, not that the result is correct. Testing a program with known inputs and expected results is critical in verifying functionality.

This appendix has focused on using C on a system with a console, such as a Linux computer. Section 8.6 describes how C is used to program PIC32 microcontrollers used in embedded systems. Microcontrollers are usually programmed in C because the language provides nearly as much low-level control of the hardware as assembly language, yet is much more succinct and faster to write.

This page intentionally left blank

Further Reading

Berlin L., *The Man Behind the Microchip: Robert Noyce and the Invention of Silicon Valley*, Oxford University Press, 2005.

The fascinating biography of Robert Noyce, an inventor of the microchip and founder of Fairchild and Intel. For anyone thinking of working in Silicon Valley, this book gives insights into the culture of the region, a culture influenced more heavily by Noyce than any other individual.

Colwell R., *The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips*, Wiley, 2005.

An insider's tale of the development of several generations of Intel's Pentium chips, told by one of the leaders of the project. For those considering a career in the field, this book offers views into the managment of huge design projects and a behind-the-scenes look at one of the most significant commercial microprocessor lines.

Ercegovac M., and Lang T., *Digital Arithmetic*, Morgan Kaufmann, 2003.

The most complete text on computer arithmetic systems. An excellent resource for building high-quality arithmetic units for computers.

Hennessy J., and Patterson D., *Computer Architecture: A Quantitative Approach, 5th ed.*, Morgan Kaufmann, 2011.

The authoritative text on advanced computer architecture. If you are intrigued about the inner workings of cutting-edge microprocessors, this is the book for you.

Kidder T., *The Soul of a New Machine*, Back Bay Books, 1981.

A classic story of the design of a computer system. Three decades later, the story is still a page-turner and the insights on project managment and technology still ring true.

Pedroni V., *Circuit Design and Simulation with VHDL, 2nd ed.*, MIT Press, 2010.

A reference showing how to design circuits with VHDL.

Ciletti M., *Advanced Digital Design with the Verilog HDL, 2nd ed.*, Prentice Hall, 2010.

A good reference for Verilog 2005 (but not SystemVerilog).

SystemVerilog IEEE Standard (IEEE STD 1800).

The IEEE standard for the SystemVerilog Hardware Description Language; last updated in 2009. Available at *ieeexplore.ieee.org*.

VHDL IEEE Standard (IEEE STD 1076).

The IEEE standard for VHDL; last updated in 2008. Available from IEEE. Available at *ieeexplore.ieee.org*.

Wakerly J., *Digital Design: Principles and Practices, 4th ed*., Prentice Hall, 2006.

A comprehensive and readable text on digital design, and an excellent reference book.

Weste N., and Harris D., *CMOS VLSI Design, 4th ed*., Addison-Wesley, 2011.

Very Large Scale Integration (VLSI) Design is the art and science of building chips containing oodles of transistors. This book, coauthored by one of our favorite writers, spans the field from the beginning through the most advanced techniques used in commercial products.

Index

Page numbers in *italics* indicate figures, tables and text boxes.

0, 22. *See also* LOW, OFF
1, 22. *See also* HIGH, ON
32-bit datapath, 461
32-bit microprocessor, 454
4004 microprocessor chip, 458, *459*
74xx series logic, 583–587
  parts,
    2:1 mux (74157), *586*
    3:8 decoder (74138), *586*
    4:1 mux (74153), *586*
    AND (7408), *585*
    AND3 (7411), *585*
    AND4 (7421), *585*
    counter (74161, 74163), *586*
    FLOP (7474), 583, *585*
    NAND (7400), *585*
    NOR (7402), *585*
    NOT (7404), 583
    OR (7432), *585*
    register (74377), *586*
    tristate buffer (74244), *586*
    XOR (7486), *585*
80386 microprocessor chip, 459, *460*
80486 microprocessor chip, 460, *461*
`#define`, 627–628
`#include`, 628–629. *See also* Standard libraries

A

Abstraction, 4–5
  digital. *See* Digital abstraction
Accumulator, 353
Acquisition time. *See* Sampling time
Active low, 74–75
A/D conversion, 531–533
  registers in, 532
ADCs. *See* Analog/digital converters
`add`, 297
Adders, 239–246
  carry-lookahead, 241
  carry propagate, 240
  full, *56*, 240
  half, 240
  HDL for, 184, 200
  prefix, 243
  ripple-carry, 240
`addi`, 304
`addiu`, 345
`addu`, 345
Addition, 14–15, 17–18, 235, 239–246, *297*. *See also* Adders
  binary, 14–15
  floating point, 258–259
  MIPS instructions, 344–345, 622
  signed binary, 15–17
Address. *See also* Memory
  physical, 497–501
  translation, 497–500
  virtual, 497. *See also* Virtual memory
Addressing modes
  MIPS, 333–335
    base, 333
    immediate, 333
    PC-relative, 333–334
    pseudo-direct, 334–335
    register-only, 333
  x86, 349
Advanced Micro Devices (AMD), 296, 375, 457, 460
Advanced microarchitecture, 444–458
  branch prediction. *See* Branch prediction
  deep pipelines. *See* Deep pipelines
  heterogeneous multiprocessors. *See* Heterogeneous multiprocessors
  homogeneous multiprocessors. *See* Homogeneous multiprocessors
  multithreading. *See* Multithreading
  out-of-order processor. *See* Out-of-order processor
  register renaming. *See* Register renaming
  single instruction multiple data. *See* Single Instruction Multiple Data
  superscalar processor. *See* Superscalar processor
Altera FPGA, 274–279
ALU. *See* Arithmetic/logical unit
ALU decoder, 382–384
  HDL for, 432
ALUControl, 378, 384
ALUOp, 382–384
ALUResult, 378
ALUSrc, 384
ALUSrcA, 397
ALUSrcB, 397
AMAT. *See* Average memory access time
AMD. *See* Advanced Micro Devices
Amdahl, Gene, 480

- Amdahl's Law, 480
- American Standard Code for Information Interchange (ASCII), 322, *323*, 630, 649–650
- Analog I/O, 531–537
  - A/D conversion, 532–533
  - D/A conversion, 533–537
  - Pulse-width modulation (PWM), 536–537
- Analog-to-digital converters (ADCs), 531–533
- Analytical engine, 7, *8*
- AND gate, 20–22, 179
  - chips (7408, 7411, 7421), 585
  - truth table, *20*, *22*
  - using CMOS transistors, 32–33
- `and`, 311
- `andi`, 311–312
- AND-OR (AO) gate, 46
- Anode, 27
- Application-specific integrated circuits (ASICs), 591
- Architectural state, 310, 371–372
- Architecture, 295–356, 619–622
  - MIPS
    - addressing modes, 333–335
    - assembly language, 296–304, 619–622
    - instructions, 619–622
    - machine language, 305–310
    - operands, 298–304
  - x86, 347–356
- Arguments, 326, 332–333, 637
  - pass by reference, 644
  - pass by value, 644
- Arithmetic
  - C operators, 633–635
  - circuits, 239–254
  - HDL operators, 185
  - MIPS instructions, 310–314
  - packed, 454
- Arithmetic/logical unit (ALU), 248–250, 378
  - implementation of, *249*
  - in MIPS processor, 382–385. *See also* ALUControl, ALUOp
- Arrays, 320–324, 645–651
  - accessing, 320–322, 645
  - as input argument, 646–647
  - bytes and characters, 322–324, 649–651
  - comparison or assignment of, 650
  - declaration, 645
  - indexing, 320–322, 645–649
  - initialization, 645–646
  - multi-dimension, 648–649
- ASCII. *See* American Standard Code for Information Interchange
- ASICs. *See* Application-specific integrated circuits
- Assembler, 338–339, 666
- Assembler directives, 338
- Assembler temporary register (`$at`), 342
- Assembly language, MIPS, 295–356, 619–622. *See also* MIPS instructions
  - instructions, 296–304, 619–622
  - logical instructions, 311–312
  - operands, 298–304
  - translating high-level code into, 300
  - translating machine language to, 309
  - translating to machine language, 306–307
- Assembly language, x86. *See* x86 instructions
- Associativity
  - in Boolean algebra, 62, 63
  - in caches, 481, 486–488
- Astable circuits, 119
- Asymmetric multiprocessors. *See* Heterogeneous multiprocessors
- Asynchronous circuits, 120–123
- Asynchronous resettable flip-flops
  - definition, 116
  - HDL, 194–196
- Asynchronous serial link, *522*. *See also* Universal Asynchronous Receiver Transmitter (UART)
- AT Attachment (ATA), 562
- Average memory access time (AMAT), 479, 492

B

- Babbage, Charles, *7*
- Base address, 301–302, 307, 320–322, 324
- Base addressing, 333
- Baud rate register (BRG), 518
- BCD. *See* Binary coded decimal
- Behavioral modeling, 173–174
- Benchmarks, 375
- `beq`, 314–315
- Biased exponent, 257
- Big-endian memory, 302–303
- Big-endian order, 178
- Binary addition, 14–15. *See also* Adders, Addition
- Binary coded decimal (BCD), 258
- Binary encoding, 125–126, 129–131
  - for divide-by-3 counter, 129–131
  - for traffic light FSM, 125–126
- Binary numbers
  - signed, 15–19
  - unsigned, 9–11
- Binary to decimal conversion, 10, *10–11*
- Binary to hexadecimal conversion, 12
- Bipolar junction transistors, 26
- Bipolar motor drive, *555*
- Bipolar signaling, 524
- Bipolar stepper motor, 554, *554–555*
  - AIRPAX LB82773-M1, 554, *555*
  - direct drive current, *556*
- Bistable element, 109
- Bit, 8
  - dirty, 494
  - least significant, 13, *14*
  - most significant, 13, *14*
  - sign, 16
  - use, 490
  - valid, 484
- Bit cells, 264–269
  - DRAM, 266–267
  - ROM, 268–269
  - SRAM, 267
- Bitline, 264
- Bit swizzling, 188
- Bitwise operators, 177–179
- Block, 481
- Block offset, 488–489
- Block size (*b*), 481, 488–489
- Blocking and nonblocking assignments, 199–200, 205–209
- BlueSMiRF silver module, 548, *548*
- Bluetooth wireless communication, 547–548
  - BlueSMiRF silver module, 548
  - classes, *547*
  - PIC32 to PC link, *548*
- `bne`, 314–315
- Boole, George, *8*
- Boolean algebra, 60–66
  - axioms, 61

  - equation simplification, 65–66
  - theorems, 61–64
- Boolean equations, 58–60
  - product-of-sums (POS) form, 60
  - sum-of-products (SOP) form, 58–60
- Boolean logic, 8. *See also* Boolean algebra, Logic gates
- Boolean theorems, 61–64
  - associativity, 63
  - combining, 62
  - complements, 62
  - consensus, 62, 64
  - covering, 62
  - De Morgan's, 63–64
  - distributivity, 63
  - idempotency, 62
  - identity, 62
  - involution, 62
  - null element, 62
- *Branch*, 384
- Branch equal (`beq`)
  - machine code for, 334
  - processor implementations of, 381–382, 395–396, 401–402
- Branch hazards. *See* Control hazards
- Branch misprediction penalty, 421–422
- Branch prediction, 446–447
- Branch target address (BTA), 333–334, 381
- Branch target buffer, 446
- Branching
  - conditional, 314–315
  - unconditional (jump), 315–316
- Breadboards, 600–601
- BTA. *See* Branch target address
- Bubble, 20, 63, 419
  - pushing, 63–64, 71–73
- Buffers, 20
  - lack of, 117
  - tristate, 74–75
- Bugs, 175
  - in C code, 667–671
- Bus, 56
  - tristate, 75
- Bypassing, 416. *See also* Forwarding
- Byte, 13–14, 322–324. *See also* Character
  - least significant, 13–14
  - most significant, 13–14
- Byte-addressable memory, 301–303
  - big-endian, 302–303
  - little-endian, 302–303
- Byte offset, 483

C

- C programming, 623–671
  - common mistakes. *See* Common mistakes
  - compilation. *See* Compilation
  - conditional statements. *See* Conditional statements
  - control-flow statements. *See* Control-flow statements
  - data types. *See* Data types
  - function calls. *See* Function calls
  - loops. *See* Loops
  - operators. *See* Operators
  - running, 626
  - simple program, 625–626
  - standard libraries. *See* Standard libraries
  - variables. *See* Variables
- Caches, 480–495
  - address fields,
    - block offset, 488–489
    - byte offset, 483
    - set bits, 483
    - tag, 483
  - advanced design, 491–495
  - evolution of, in MIPS, 495
  - multiple level, 492
  - nonblocking, 566
  - organizations, 490
    - direct mapped, 482–486
    - fully associative, 487–488
    - multiway set associative, 486–487
  - parameters
    - block, 481
    - block size, 481, 488–489
    - capacity (C), 480–481
    - degree of associativity (N), 486
    - number of sets (S), 481
  - performance of
    - hit, 478–480
    - hit rate, 478–480
    - miss, 478–480, 493
      - capacity, 493
      - compulsory, 493
      - conflict, 486, 493
      - penalty, 488
    - miss rate, 478–480
      - reducing, 493–494
      - miss rate *vs.* cache parameters, *493–494*
  - replacement policy, 490–491
  - status bits
    - dirty bit (D), 494
    - use bit (U), 490
    - valid bit (V), 484
  - write policy, 494–495
    - write-back, 494–495
    - write-through, 494–495
- CAD. *See* Computer-aided design
- Callee-saved registers, 329
- Canonical form. *See* Sum-of-products, Product-of-sums
- Capacitors, 28
- Capacity, of cache, 480–481
- Capacity miss, 493
- Carry propagate adder (CPA). *See* Ripple-carry adder, Carry-lookahed adder, and Prefix adder
- Carry-lookahead adder (CLA), 241–243, *242*
- Case statement, in HDL, 201–203. *See also* Switch/case statement
- Casez, case?, in HDL, 205
- Cathode, 27
- Cathode ray tube (CRT), 541–542. *See also* VGA monitor
  - horizontal blanking interval, 542
  - vertical blanking interval, 542
- Cause register, 343–344, 441
- Character LCDs, 538–541
- Characters (`char`), 322–324, 630, 649
  - arrays, 322–324. *See also* Strings
  - C type, 649
- Chips, 28
  - multiprocessors, 456
- Chopper constant current drive, 556
- Circuits
  - 74xx series. *See* 74xx series logic
  - application-specific integrated (ASICs), 591
  - astable, 119
  - asynchronous, 120, 122–123
  - combinational. *See* Combinational logic
  - definition of, 55
  - delay, 88–92
  - multiple-output, 68
  - priority, 68
  - sequential. *See* Sequential logic

- Circuits (*Continued*)
  - synchronous, 122–123
  - synchronous sequential, 120–123, *122*
  - synthesized, *176*, *179*, *181*
  - timing, 88–95
  - with two-stage pipeline, *160*
  - without glitch, *95*
- CISC. *See* Complex Instruction Set Computer
- CLBs. *See* Configurable logic blocks
- Clock cycles per instruction (CPI), 444, 446
- Clock period, 142, 376
- Clock skew, 148–151
- Clustered multiprocessing, 456
- CMOS. *See* Complementary Metal-Oxide-Semiconductor Logic
- Combinational composition, *56*
- Combinational logic, 174
  - design, 55–106
    - Boolean algebra, 60–66
    - Boolean equations, 58–60
    - building blocks, 83–88, 239–254
    - delays, 88–92
    - don't cares, 81–82
    - Karnaugh maps (K-maps), 75–83
    - multilevel, 66–73
    - precedence, 58
    - timing, 88–95
    - two-level, 69
    - X's (contention). *See* Contention
    - X's (don't cares). *See* Don't cares (X)
    - Z's (floating). *See* Floating (Z)
  - HDLs and. *See* Hardware description languages
  - truth tables with don't cares, 69, 81–82, 205
- Combining theorem, 62
- Command line arguments, 666–667
- Comments
  - in C, 627
  - in MIPS assembly, 297
  - in SystemVerilog, 180
  - in VHDL, 180
- Common mistakes in C, 667–671
- Comparators, 246–248
- Comparison
  - in hardware. *See* Comparators, ALU
  - in MIPS assembly, 319–320, 345
  - using ALU, 250
- Compilation, in C, 626–627, 665–666
- Compiler, 338–339
  - for C, 626–627, 665–666
- Complementary Metal-Oxide-Semiconductor Logic (CMOS), 26–34
- Complements theorem, 62
- Complex instruction set computer (CISC), 298, 347
- Complexity management, 4–7
  - digital abstraction, 4–5
  - discipline, 5–6
  - hierarchy, 6–7
  - modularity, 6–7
  - regularity, 6–7
- Compulsory miss, 493
- *Computer Architecture* (Hennessy & Patterson), 444
- Computer-aided design (CAD), 71, 129
- Concurrent signal assignment statement, 179, 183–184, 193, 200–206
- Condition codes. *See* Status flags
- Conditional assignment, 181–182
- Conditional branches, 314–315
- Conditional operator, 181–182
- Conditional signal assignments, 181–182
- Conditional statements
  - in C, 639–640
    - if, 639–640
    - if/else, 639
    - switch/case, 639–640
  - in HDL, 194, 201–205
    - case, 201–203
    - casez, case?, 205
    - if, if/else, 202–205
  - in MIPS assembly, 316–317
    - if, 316–317
    - if/else, 317
    - switch/case, 317
- Configurable logic blocks (CLBs), 274, 589. *See also* Logic elements
- Conflict miss, 493
- Consensus theorem, 62, 64
- Constants
  - in C, 627–628
  - in MIPS assembly, 304, 313. *See also* Immediates
- Contamination delay, 88–92. *See also* Short path
- Contention (X), 73–74
- Context switching, 455
- Continuous assignment statements, 179, 193, 200, 206
- Control hazards, 415, 421–424
- Control signals, 91, 249
- Control unit. *See also* ALU decoder, Main decoder
  - of multicycle MIPS processor, 396–408
  - of pipelined MIPS processor, 413–414
  - of single-cycle MIPS processor, 382–387
- Control-flow statements
  - conditional statements. *See* Conditional statements
  - loops. *See* Loops
- Coprocessor 0 registers, 441. *See also* Cause and EPC
- Core Duo microprocessor chip, 464
- Core i7 microprocessor chip, *465*
- Cores, 456
- Counters, 260
  - divide-by-3, *130*
- Covering theorem, 62
- CPA. *See* Carry propagate adder (CPA)
- CPI. *See* Clock cycles per instruction, Cycles per instruction
- Critical path, 89–92, 388
- Cross-coupled inverters, 109, *110*
  - bistable operation of, *110*
- CRT. *See* Cathode ray tube
- Cycle time. *See* Clock period
- Cycles per instruction (CPI), 375
- Cyclic paths, 120
- Cyclone IV FPGA, 274–279

D

- D flip-flops. *See* flip-flops
- D latch. *See* Latch
- D/A conversion, 533–537
- DACs. *See* Digital-to-analog converters
- Data Acquisition Systems (DAQs), 562–563
  - myDAQ, 563
- Data hazards, 415–421
- Data memory, 373
  - HDL for, 439
- Data segment, 340
- Data sheets, 591–596

Data types, 643–657
  arrays. *See* Arrays
  characters. *See* Character (`char`)
  dynamic memory allocation. *See* Dynamic memory allocation (`malloc` and `free`)
  linked list. *See* Linked list
  pointers. *See* Pointers
  strings. *See* Strings (`str`)
  structures. *See* Structures (`struct`)
  typedef, 653–654
Datapath
  multicycle MIPS processor, 390–396
  pipelined MIPS processor, 412–413
  single-cycle MIPS processor, 376–382
DC. *See* Direct current
DC motors, 548–552, *549*
  H-bridge, 549, *550*
  shaft encoder, 549–552
DC transfer characteristics, 24–26. *See also* Noise margins
DDR3. *See* Double-data rate memory
DE-9 cable, *524*
De Morgan's theorem, 63
Decimal numbers, 9
Decimal to binary conversion, 11
Decimal to hexadecimal conversion, 13
Decode stage, 409–411
Decoders
  definition of, 86–87
  HDL for
    behavioral, 202–203
    parameterized, 219
  logic using, 87–88
  Seven-segment. *See* Seven-segment display decoder
Deep pipelines, 444–445
Delay generation using counters, 528–529
`Delaymicros` function, 528
Delays, logic gates. *See* Propagation delay
  in HDL (simulation only), 188–189
`DeleteUser` function, 655
De Morgan, Augustus, *63*
De Morgan's theorem, 63–64
Dennard, Robert, *266*
Destination register (`rd` or `rt`), 378–379, 385, 393
Device driver, 507–508, 526
Device under test (DUT), 220
Dice, 28
Dielectric, 28
Digital abstraction, 4–5, 7–9, 22–26
Digital circuits. *See* Logic
Digital signal processor (DSP), 457
Digital system implementation, 583–617
  74xx series logic. *See* 74xx series logic
  application-specific integrated circuits (ASICs), 591
  assembly of, 599–602
  breadboards, 600–601
  data sheets, 591–596
  economics, 615–617
  logic families, 597–599
  packaging, 599–602
  printed circuit boards, 601–602
  programmable logic, 584–591
Digital-to-analog converters (DACs), 531
DIMM. *See* Dual inline memory module
Diodes, 27–28
  p-n junction, *28*
DIPs. *See* Dual-inline packages
Direct current (DC) transfer characteristics, 24, *25*
Direct mapped cache, 482–486, *484*
Direct voltage drive, 554
Dirty bit (*D*), 494
Discipline
  dynamic, 142–151. *See also* Timing analysis
  static, 142–151. *See also* Noise margins
Discrete-valued variables, 7
Distributivity theorem, 63
`div`, 314
Divide-by-3 counter
  design of, 129–131
  HDL for, 210–211
Divider, 253–254
Division
  circuits, 253–254
  MIPS instruction, 314
  MIPS signed and unsigned instructions, 345
`divu`, 345
Don't care (X), 69, 81–83, 205
Dopant atoms, 27
Double, C type, 630–631
Double-data rate memory (DDR), 266, 561
Double-precision formats, 257–258
Do/while loops, in C, 641–642
DRAM. *See* Dynamic random access memory
Dual inline memory module (DIMM), 561
Dual-inline packages (DIPs), *28*, 583, 599
Dynamic branch predictors, 446
Dynamic data segment, 337
Dynamic discipline, 142–151. *See also* Timing analysis
Dynamic memory allocation (`malloc`, `free`), 654–655
  in MIPS memory map, 337
Dynamic power, 34
Dynamic random access memory (DRAM), 266, 267, 475–478, 561

E

Economics, 615
Edge-triggered flip-flop. *See* flip-flop
EEPROM. *See* Electrically erasable programmable read only memory
EFLAGS register, 350
Electrically erasable programmable read only memory (EEPROM), 269
Embedded I/O (input/output) systems, 508–558
  analog I/O, 531–537
    A/D conversion, 532–533
    D/A conversion, 533–536
  digital I/O, 513–515
  general-purpose I/O (GPIO), 513–515
  interrupts, 529–531
  LCDs. *See* Liquid Crystal Displays
  microcontroller peripherals, 537–558
  motors. *See* Motors
  PIC32 microcontroller, 509–513
  serial I/O, 515–527. *See also* Serial I/O
  timers, 527–529
  VGA monitor. *See* VGA monitor, 493
Enabled flip-flops, 115–116
Enabled registers, 196–197. *See also* flip-flops
`EPC`. *See* Exception program counter
EPROM. *See* Erasable programmable read only memory

Equality comparator, 247
Equation minimization
    using Boolean algebra, 65–66
    using Karnaugh maps. *See* Karnaugh maps
Erasable programmable read only memory (EPROM), 269, 588
Ethernet, 561
Exception program counter (`EPC`), 343–344
Exceptions, 343–344, 440–443
    `Cause`. *See* Cause register
    cause codes, 344
    `EPC`. *See* Exception program counter
    handler, 343
    processor support for, 440–443
        circuits, 441–442
        controller, 442–443
Executable file, 340
Execution time, 375
`exit`, 663
Extended instruction pointer (EIP), 348

F

Factorial function call, 330–331
    stack during, *331*
Factoring state machines, 134–136
FDIV. *See* Floating-point division
Field programmable gate arrays (FPGAs), 274–279, 457, 520, 543, 564, 589–591
    driving VGA cable, *543*
    in SPI interface, 519–521
File manipulation, in C, 660–662
Finite state machines (FSMs), 123–141, 209–213
    deriving from circuit, 137–140
    divide-by-3 FSM, 129–131, 210–211
    factoring, 134–136, *136*
    in HDL, 209–213
    LE configuration for, 277–278
    Mealy FSM, 132–134
    Moore FSM, 132–134
    multicycle control, 396–408, *405*, *408*
    snail/pattern recognizer FSM, 132–134, 212–213
    state encodings, 129–131. *See also* Binary encoding, One-cold encoding, One-hot encoding
    state transition diagram, 124, *125*
    traffic light FSM, 123–129
Fixed-point numbers, 255–256
Flags, 250
Flash memory, 269. *See also* Solid state drives
Flip-flops, 114–118, 193–197. *See also* Registers
    back-to-back, *145*, 152–157, *197*. *See also* Synchronizer
    comparison with latches, 118
    enabled, 115–116
    HDL for, 436. *See also* Registers
    metastable state of. *See* Metastability
    register, 114–115
    resettable, 116
    scannable, 262–263
    shift register, 261–263
    transistor count, 114, 117
    transistor-level, 116–117
Float, C type, 628–631
    print formats of, 658–659
Floating output node, 117
Floating point division (FDIV) bug, 175
Floating (Z), 74–75
    in HDLs, 186–188
Floating-gate transistor, 269. *See also* Flash memory
Floating-point coprocessors, 457
Floating-point division (FDIV), *259*
Floating-point instructions, MIPS, 346–347
Floating-point numbers, 256–257
    addition, 258–259
    formats, single- and double-precision, 256–258
    in programming. *See* Float and Double
    rounding, 258
    special cases
        infinity, 257
        NaN, 257
Floating-point unit (FPU), *259*, 461
For loops, 319–320, 322, 642
Format conversion (`atoi`, `atol`, `atof`), 663–664
Forwarding, 416–418. *See also* Hazards
FPGAs. *See* Field programmable gate arrays
FPU. *See* Floating-point unit
Frequency shift keying (FSK), *548*
    and GFSK waveforms, *548*
Front porch, 542
FSK. *See* Frequency Shift Keying
FSMs. *See* Finite state machines
Full adder, 56, 182, 184, 200, 240
    using always/process statement, 200
Fully associative cache, 487–488
Funct field, 305, 621–622
Function calls, 325–333, 637–638
    arguments, 325–326, 637
    leaf, 330
    naming conventions, 638
    nonleaf, 330
    preserved and non-preserved registers, 329–332
    prototypes, 638
    recursive, 330–332
    return, 325–326, 637
    stack, use of, 327–333. *See also* Stack
    with no inputs or outputs, 325, 637
Fuse-programmable ROM, 269

G

Gated time accumulation, *529*
Gates
    AND, 20, 22, *128*
    buffer, 20
    multiple-input, 21–22
    NAND, 21, 31
    NOR, 21–22, 111, *128*
    NOT, 20
    OR, 21
    transistor-level. *See* Transistors
    XNOR, 21
    XOR, 21
General-purpose I/O (GPIO), 513
    PIC32 ports (pins) of, *515*
    switches and LEDs example, 513–514
Generate signal, 241, 243
Genwaves function, 535
Glitches, 92–95
Global data segment, 336–337
Global pointer (`$gp`), 337
GPIO. *See* General-purpose I/O
Graphics accelerators, 464
Graphics processing unit (GPU), 457

- Gray codes, 76
- Gray, Frank, *76*
- Ground (GND), 22
  - symbol for, 31

H

- Half adder, 240, *240*
- Hard disk, 478–479. *See also* Hard drive
- Hard drive, 478–479, 496. *See also* Hard disk, Solid state drive, and Virtual memory
- Hardware description languages (HDLs). *See also* SystemVerilog, VHDL
  - capacity, 493
  - combinational logic, 174, 198
    - bitwise operators, 177–179
    - blocking and nonblocking assignments, 205–209
    - case statements, 201–202
    - conditional assignment, 181–182
    - delays, 188–189
  - data types, 213–217
  - history of, 174–175
  - if statements, 202–205
    - internal variables, 182–184
    - numbers, 185
    - operators and precedence, 184–185
    - reduction operators, 180–181
  - modules, 173–174
  - parameterized modules, 217–220
  - processor building blocks, 434–437
  - sequential logic, 193–198, 209–213
  - simulation and synthesis, 175–177
  - single-cycle MIPS processor, 429–440
  - structural modeling, 190–193
  - testbench, 220–224, 437–438
- Hardware handshaking, 523
- Hardware reduction, 70–71. *See also* Equation minimization
- Hazard unit, 416–427
- Hazards. *See also* Hazard unit
  - control hazards, 415, 421–424
  - data hazards, 416–421
  - read after write (RAW), 415, 451
  - solving
    - control hazards, 421–424
    - forwarding, 416–418
    - stalls, 418–421
  - write after read (WAR), 451
  - write after write (WAW), 451–452
- H-bridge control, *550*
- HDL. *See* Hardware description language, SystemVerilog, and VHDL
- Heap, 337
- Heterogeneous multiprocessors, 456–458
- Hexadecimal numbers, 11–13
- Hexadecimal to binary and decimal conversion, 11, *12*
- Hierarchy, 6
- HIGH, 22. *See also* 1, ON
- High-level programming languages, 296, 624
  - compiling, assembling, and loading, 336–341
  - translating into assembly, 300
- High-performance microprocessors, 444
- Hit, 478
- Hit rate, 478–480
- Hold time constraint, 142–148
  - with clock skew, 149–151
- Hold time violations, 145, *146*, 147–148, 150–151
- Homogeneous multiprocessors, *456*
- Hopper, Grace, *337*

I

- IA-64, *354*
- IA-32 architecture. *See* x86
- ICs. *See* Integrated circuits
- Idempotency theorem, 62
- Identity theorem, 62
- Idioms, 177
- If statements
  - in C, 639
  - in HDL, 202–205
  - in MIPS assembly, 316–317
- If/else statements, 317, 649
  - in C, 639–640
  - in HDL, 202–205
  - in MIPS assembly, 317
- IM. *See* Instruction memory
- Immediate addressing, 333
- Immediates, 304, 313. *See also* Constants
  - 32-bit, 313
  - immediate field, 307–308
  - logical operations with, *311*
- Implicit leading one, 256
- Information, amount of, 8
- Initializing
  - arrays in C, 645–646
  - variables in C, 633
- InitTimer1Interrupt function, 530
- Input/output elements (IOEs), 274
- Input/Output (I/O) systems, 506–569
  - device driver, 507–508, 526
  - embedded I/O systems. *See* Embedded I/O systems
  - I/O registers, 507–508
  - memory-mapped I/O, 507–508
  - personal computer I/O systems. *See* Personal computer I/O systems
- Institute of Electrical and Electronics Engineers (IEEE), *257*
- Instruction encoding, x86, 352–354, *353*
- Instruction formats, MIPS
  - F-type, 346
  - I-type, 307–308
  - J-type, 308
  - R-type, 305–306
- Instruction formats, x86, 352–354
- Instruction level parallelism (ILP), 452, 455
- Instruction memory (IM), 373, 411
  - MIPS, *440*
- Instruction register (IR), 391, 398
- Instruction set, 295, 371–372. *See also* Architecture
- Instructions, MIPS, 295–347, 619–622
  - arithmetic, 299–300, 304, 314, 344–345, *620–622*
  - branching, 314–316
  - floating-point, 346–347, *622*
  - for accessing memory. *See* Loads, Stores
  - F-type, 346
  - I-type, 307–308
  - J-type, 308
  - logical, 308, 310–313
  - multiplication and division, 314, 345
  - R-type, 305–306, *621–622*
  - set less than, 319–320, 345
  - signed and unsigned, 344–345
- Instructions, x86, 347–355
- Instructions per cycle (IPC), 375
- Integer Execution Unit (IEU), 461
- Integrated circuits (ICs), 599
- Intel. *See* x86
- Intel x86. *See* x86

- Interrupt service routine (ISR), *529*. *See also* Exceptions
- Interrupts, 343, 529–531
  - PIC32, 529–531
- Invalid logic level, 186
- Inverters, 20, *119*, *178*. *See also* NOT gate
  - cross-coupled, 109, *110*
  - in HDL, 178, 199
- *An Investigation of the Laws of Thought* (Boole), *8*
- Involution theorem, 62
- I/O. *See* Input/output systems
- IOEs. *See* Input/output elements
- *IorD*, 393, 397
- IPC. *See* Instructions per cycle
- IR. *See* Instruction register
- *IRWrite*, 391, 397
- ISR. *See* Interrupt service routine
- I-type instructions, 307–308

J

- `j`, 315–316
- `jal`, 325
- Java, 322. *See also* Language
- `jr`, 315–316, 325
- JTA. *See* Jump target address
- J-type instructions, 308
- Jump, MIPS instruction, 315–316
- Jump, processor implementation, 386–387, 404–408
- Jump target address (JTA), 334–335, 386

K

- Karnaugh maps (K-maps), 75–84, 93–95, 126
  - logic minimization using, 77–83
  - prime implicants, 65, 77–81, 94–95
  - seven-segment display decoder, 79–81
    - with “don’t cares,” 81–82
  - without glitches, *95*
- Karnaugh, Maurice, *75*
- Kilobit (Kb/Kbit), 14
- Kilobyte (KB), 14
- K-maps. *See* Karnaugh maps

L

- LAB. *See* Logic array block
- Land grid array, 558
- Language. *See also* Instructions
  - assembly, 296–304
  - machine, 305–310
  - mnemonic, 297
  - translating assembly to machine, 306
- Last-in-first-out (LIFO) queue, 327. *See also* Stack
- Latches, 111–113
  - comparison with flip-flops, 109, 118
  - D, 113, *120*
  - SR, 111–113, *112*
  - transistor-level, 116–117
- Latency, 157–160, 409–411, 418
- Lattice, silicon, 27
- `lb`, load byte. *See* Loads
- `lbu`, load byte unsigned. *See* Loads
- LCDs. *See* Liquid crystal displays
- Leaf function, 330
- Leakage current, 34
- Least recently used (LRU) replacement, 490–491
  - two-way associative cache with, 490–491, *491*
- Least significant bit (lsb), 13, *14*
- Least significant byte (LSB), 13, *14*, 302
- LEs. *See* Logic elements
- Level-sensitive latch. *See* D latch
- `lh`, load half. *See* Loads
- `lhu`, load half unsigned. *See* Loads
- LIFO. *See* Last-in-first-out queue
- Line options, compiler and command, 665–667
- Linked list, 655–656
- Linker, 340–341
- Liquid crystal displays (LCDs), 538–541
- Literal, 58, 96
- Little-endian memory, 302–303, *302*
- Little-endian bus order in HDL, 178
- Loads, 345
  - base addressing of, 333
  - load byte (`lb` or `lbu`), 304, 323–324, 345
  - load half (`lh` or `lhu`), 345
  - load word (`lw`), 301–304
- Local variables, 332–333
- Locality, 476
- Logic
  - bubble pushing, 71–73
  - combinational. *See* Combinational logic
  - families, 597–599
  - gates. *See* Gates
  - hardware reduction. *See* Equation simplification and Hardware reduction
  - multilevel. *See* Multilevel combinational logic
  - programmable, 584–591
  - sequential. *See* Sequential logic
  - transistor-level. *See* Transistors
  - two-level, 69
- Logic array block (LAB), 275
- Logic arrays, 272–280. *See also* Programmable logic arrays and Field programmable gate arrays
  - transistor-level implementation, 279–280
- Logic elements (LEs), 274–279
  - of Cyclone IV, 276
  - functions built using, 277–278
- Logic families, 25, 597–599
  - compatibility of, 26
  - logic levels of, *25*
  - specifications, *597*, *599*
- Logic gates, 19–22, *179*, 584
  - AND. *See* AND gate
  - AND-OR (AO) gate, 46
  - multiple-input gates, 21–22
  - NAND. *See* NAND gate
  - NOR. *See* NOR gate
  - OR. *See* OR gate
  - OR-AND-INVERT (OAI) gate, 46
  - with delays in HDL, *189*
  - XNOR. *See* XNOR gate
  - XOR. *See* XOR gate
- Logic levels, 22–23
- Logic simulation, 175–176
- Logic synthesis, 176–177, *176*
- Logical instructions, 311–312

Logical shifter, 250
Lookup tables (LUTs), 270, 275
Loops, 317–319, 641–642
  in C,
    do/while, 641–642
    for, 642
    while, 641
  in MIPS assembly,
    for, 319–320
    while, 318–319
LOW, 22. *See also* 0, OFF
Low Voltage CMOS Logic (LVCMOS), 25
Low Voltage TTL Logic (LVTTL), 25
LRU. *See* Least recently used replacement
LSB. *See* Least significant byte
lsb. *See* Least significant bit
`lui`, load upper immediate, 313
LUTs. *See* Lookup tables
LVCMOS. *See* Low Voltage CMOS Logic
LVTTL. *See* Low Voltage TTL Logic
`lw`, load word. *See* Loads

M

Machine code, assembly and, *437*
Machine language, 305–310
  formats, 305–308
    F-type, 346
    I-type, 307–308, *307*
    J-type, 308, *308*
    R-type, 305–306, *305*
  interpreting, 308–309
  stored program, 309–310, *310*
  translating assembly language to, 306
Magnitude comparator, 247
Main decoder, 382–387
  HDL for, 432
`main` function in C, 625
Main memory, 478
Malloc function, 654
Mantissa, 258–259
Mapping, 482
Master latch, 114
Master-slave flip-flop, 114
Masuoka, Fujio, *269*
math.h, C library, 664–665
Max-delay constraint. *See* Setup time constraint
Maxterms, 58
MCM. *See* Multichip module
Mealy machines, 123, *123*, 132
  state transition and output table, *134*
  state transition diagrams, *133*
  timing diagrams for, *135*
Mean time between failure (MTBF), 153–154
Medium-scale integration (MSI) chips, 584
Memory. *See also* Memory arrays
  addressing modes, *349*
  area and delay, 266–267
  arrays. *See* Memory arrays average memory access time, 479
  big-endian, 178, 302–303
  byte-addressable, 301–303
  HDL for, 270–272
  hierarchy, *478*
  little-endian, 178, 302–303
  logic using, 270–272
  main, 478
  operands in, 301–304
  physical, 497
  ports, 265
  protection, 503. *See also* Virtual memory
  types, 265–270
    DDR, 267
    DRAM, 266
    flash, 269–270
    register file, 267–268
    ROM, 268–270
    SRAM, 266
  virtual, 478. *See also* Virtual memory
Memory arrays, 263–272. *See also* Memory
  bit cell, 264–269
  HDL for, 270–272
  logic using, 270–272
  organization, 263–265
Memory hierarchy, 478–479
Memory interface, 475–476
Memory map
  MIPS, 336–337, 341, 507
  PIC32, 509–510
Memory Performance. *See* Average Memory Access Time
Memory protection, 503
Memory systems, 475
  MIPS, 495
  performance analysis, 479–480
  x86, 564–568
Memory-mapped I/O
  address decoder, 507
  communicating with I/O devices, 507–508
  hardware, 508
Mem Write, 379, *397*
MemtoReg, 380, *397*
Metal-oxide-semiconductor field effect transistors (MOSFETs), 26
  switch models of, *30*
Metastability, 151–157
  metastable state, 110, 151
  resolution time, 151–152, 154–157
  synchronizers, 152–154
`mfc0`. *See* Move from coprocessor 0
Microarchitecture, 351–466. *See also* Architecture
  advanced. *See* Advanced microarchitecture
  architectural state. *See* Architectural state
  description of, 371–374
  design process, 372–374
  HDL representation, 429–440
  multicycle processor. *See* Multicycle MIPS processor
  performance analysis, 374–376. *See also* Performance analysis
  pipelined processor. *See* Pipelined MIPS processor
  single-cycle processor. *See* Single-cycle MIPS processor
  x86, 458–465
    evolution of, 458
Microchip ICD3, *513*
Microchip In Circuit Debugger 3 (ICD3), *513*
Microcontroller, 508
  PIC32 (PIC32MX675F512H), 509–513, *510*
    64-pin TQFP package in, *511*
    operational schematic of, *512*
    to PC serial link, *526*
    pinout of, *511*
    virtual memory map of, *510*
Microcontroller peripherals, 537–558
  Bluetooth wireless communication, 547–548
  character LCD, 538–541
    control, 540–541
    parallel interface, *539*

- Microcontroller peripherals (*Continued*)
  - motor control, 548–549
  - VGA monitor, 541–547
- Microcontroller units (MCUs), 508
- Micro-ops, 461
- Microprocessors, 3, *13*, 295
  - architectural state of, 310
  - designers, 444
  - high-performance, 444
- Millions of instructions per second, 409
- Min-delay constraint. *See* Hold time constraint
- Minterms, 58
- MIPS. *See also* Architecture and Microarchitecture
  - architecture, 296, 509
    - floating-point instructions, *346*, 346–347
  - instruction set, 385
  - microarchitectures
    - multicycle. *See* Multicycle MIPS processor
    - pipelined. *See* Pipelined MIPS processor
    - single-cycle. *See* Single-cycle MIPS processor
  - microprocessor, 441, 452, 455
    - data memory, 373
    - instruction memory, 373
    - program counter, 373
    - register file, 373
    - state elements of, *373*
  - processor control, 344
  - register set, *300*
  - *vs.* x86 architecture, *348*
- MIPS instructions, 295–356, *219–222*
  - branching. *See* Branching
  - formats
    - F-type, *622*
    - I-type, *307*, 307–308
    - J-type, 308, *308*
    - R-type, 305–307
  - multiplication and division, 314, 345
  - opcodes, *620–621*
  - R-type funct fields, *621–622*
- MIPS processors. *See* MIPS multi-cycle processor, MIPS pipelined processor, and MIPS single-cycle processor
  - HDL for. *See* MIPS single-cycle HDL
- MIPS registers,
  - co-processor 0 registers, 344, 441–443
  - program counter, 310, 372–373
  - register file, 372–373
  - register set, 298–300
- MIPS single-cycle HDL, 429–440
  - building blocks, 434–437
  - controller, 429
  - datapath, 429
  - testbench, 437–440
- Miss, 478–480, 493
  - capacity, 493
  - compulsory, 493
  - conflict, 486, 493
- Miss penalty, 488
- Miss rate, 478–480
  - and access times, *480*
- Misses
  - cache, 478
    - capacity, 493
    - compulsory, 493
    - conflict, 493
  - page fault, 497
- Modularity, 6
- Modules, in HDL
  - behavioral and structural, 173–174
  - parameterized modules, 217–220
- Moore, Gordon, *30*
- Moore machines, 123, 132
  - state transition and output table, *134*
  - state transition diagrams, *133*
  - timing diagrams for, *135*
- Moore's law, *30*
- MOS transistors. *See* Metal-oxide-semiconductor field effect transistors
- MOSFET. *See* Metal-oxide-semiconductor field effect transistors
- Most significant bit (msb), 13, *14*
- Most significant byte (MSB), 13, *14*, 302
- Motors
  - DC, 548–552
  - H-bridge, *550*
  - servo, 549, *552–554*
  - stepper, 548, *554–558*
- Move from coprocessor 0 (`mfc0`), 344, 441–443. *See also* Exceptions
- MPSSE. *See* Multi-Protocol Synchronous Serial Engine
- MSB. *See* Most significant byte
- msb. *See* Most significant bit
- MSI chips. *See* Medium-scale integration
- MTBF. *See* Mean time between failure
- `mul`, multiply, 32-bit result, 314
- `mult`, multiply, 64-bit result, 314
- Multichip module (MCM), 566
- Multicycle MIPS processor, 389–408
  - control, 396–404
  - datapath, 390–396
  - performance, 405–408
- Multilevel combinational logic, 69–73. *See also* Logic
- Multilevel page tables, 504–506
- Multiple-output circuit, 68–69
- Multiplexers, 83–86
  - definition of, 83–84
  - HDL for
    - behavioral model of, 181–183
    - parameterized N-bit, 218–219
    - structural model of, 190–193
  - logic using, 84–86
  - symbol and truth table, *83*
- Multiplicand, 252
- Multiplication, 314, 345. *See also* Multiplier
  - MIPS instruction, 314
  - signed and unsigned instructions, 345
- Multiplier, 252–253
  - schematic, 252
  - HDL for, 253
- Multiprocessors
  - chip, 456
  - heterogeneous, 456–458
  - homogeneous, 456
- Multi-Protocol Synchronous Serial Engine (MPSSE), *563*, *563*
- Multithreaded processor, 455
- Multithreading, 455
- `multu`, 345
- Mux. *See* Multiplexers
- myDAQ, 563

N

- NAND (7400), *585*
- NAND gate, 21, *21*, *31*
  - CMOS, 31–32, *31–32*
- Nested if/else statement, 640
- Nibbles, 13–14
- nMOS transistors, 28–31, *29–30*
- Noise margins, 23–26, *23*
  - calculating, 23–24
- Nonarchitectural state, 372
- Nonblocking and blocking assignments, 199–200, 205–209

- Nonleaf function, 330
- Nonpreserved registers, *329*, *330*
- `nop`, 342
- `nor`, 311
- NOR gate, 21–22, 111, *128*, *585*
  - chip (7402), *585*
  - CMOS, 32
  - pseudo-nMOS logic, *33*
  - truth table, *22*
- Not a number (NaN), 257
- NOT gate, 20
  - chip (7404), *585*
  - CMOS, 31
- Noyce, Robert, *26*
- Null element theorem, 62
- Number conversion
  - binary to decimal, *10–11*
  - binary to hexadecimal, 12
  - decimal to binary, 11, 13
  - decimal to hexadecimal, 13
  - hexadecimal to binary and decimal, 11, *12*
  - taking the two's complement, 16
- Number systems, 9–19
  - binary, 9–11, *10–11*
  - comparison of, 18–19, *19*
  - decimal, *9*
  - estimating powers of two, 14
  - fixed-point, *255*, *255–256*
  - floating-point, 256–259
    - addition, 258–259, *259*
    - special cases, 257
  - hexadecimal, 11–13, *12*
  - negative and positive, 15
  - signed, 15
  - unsigned, 9–11

O

- OFF, 23. *See also* 0, LOW
- Offset, 391, *392*
- ON, 23. *See also* 1, HIGH
- One-bit dynamic branch predictor, 446–447
- One-cold encoding, 130
- One-hot encoding, 129–131
- One-time programmable (OTP), 584
- Opcode, 305, *620–621*
- Operands
  - MIPS, 298–304
    - immediates (constants), 304, 313
    - memory, 301–304
    - registers, 298–300
  - x86, 348–350, *349*
- Operation code. *See* Opcode
- Operators
  - in C, 633–636
  - in HDL, 177–185
    - bitwise, 177–181
    - precedence, 185
    - reduction, 180–181
    - table of, 185
    - ternary, 181–182
- `or`, 311
- OR-AND-INVERT (OAI) gate, 46
- OR gate, 21
- `ori`, 311–312
- OTP. *See* One-time programmable
- Out-of-order execution, *453*
- Out-of-order processor, 450–452
- Overflow
  - handling exception for, 343–345, 440–443
  - with addition, 15
- Oxide, 28

P

- Packages, chips, 599–600
- Packed arithmetic, 454
- Page fault, 497
- Page number, 498
- Page offset, 498
- Page table, 498, 500–501
  - number, 504
  - offset, 504
- Pages, 497
- Paging, 504
- Parallel I/O, 515
- Parallelism, 157–160
- Parity gate. *See* XOR
- Partial products, 252
- Pass by reference, 644
- Pass by value, 644
- Pass gate. *See* Transmission gates
- PC. *See* Program counter or Personal computer
- PCB. *See* Printed circuit board
- PCI. *See* Peripheral Component Interconnect
- PCI express (PCIe), 560
- PC-relative addressing, 333–334
- *PCSrc*, 395, *396–397*, 397
- *PCWrite*, 393, 397
- Pentium processors, 460, *462*
  - Pentium 4, 375, *463*, 463–464
  - Pentium II, 461
  - Pentium III, 375, 461, *462*
  - Pentium M, 464
  - Pentium Pro, 461
- Perfect induction, proving theorems using, 64–65
- Performance Analysis, 374–376. *See also* Average Memory Access Time
  - multi-cycle MIPS processor, 405–407
  - pipelined MIPS processor, 426–428
  - processor comparison, 428
  - single-cycle MIPS processor, 388–389
- Periodic interrupts, 530–531
- Peripheral bus clock (PBCLK), *512*
- Peripheral Component Interconnect (PCI), 560
- Peripherals devices. *See* Input/output systems
- Personal computer (PC). *See* x86
- Personal computer (PC) I/O systems, 558–564
  - data acquisition systems, 562–563
  - DDR3 memory, 561
  - networking, 561–562
  - PCI, 560
  - SATA, 562
  - USB, 559–560, 563–564
- Phase locked loop (PLL), 544
- Physical address extension, 567
- Physical memory, 497
- Physical page number (PPN), 499
- Physical pages, 497
- PIC32 microcontroller (PIC32MX675F512H), 509–513. *See also* Embedded I/O systems
- Pipelined MIPS processor, 409–428
  - abstract view of, 411
  - control, 413–414
  - datapath, 412–413
  - description, 409–412
  - hazards, 414–426. *See also* Hazards
  - performance, 426–428
  - throughput, 411
- Pipelining, 158–160. *See also* Pipelined MIPS processor

PLAs. *See* Programmable logic arrays
Plastic leaded chip carriers (PLCCs), 599
Platters, 496
PLCCs. *See* Plastic leaded chip carriers
PLDs. *See* Programmable logic devices
PLL. *See* Phase locked loop
pMOS transistors, 28–31, *29*
Pointers, 643–645, 647, 650, 652, 654
POS. *See* Product-of-sums form
Positive edge-triggered flip-flop, 114
Power consumption, 34–35
Power processor element (PPE), 457
PPN. *See* Physical page number
Prefix adders, 243–245, *244*
Prefix tree, 245
Preserved registers, 329–330, *330*
Prime implicants, 65, 77
Printed circuit boards (PCBs), 601–602
`printf`, 657–659
Priority
    circuit, 68–69
    encoder, 102–103, 105
Procedure calls. *See* Function calls
Processor-memory gap, 477
Processor performance comparison
    multicycle MIPS processor, 407–408
    pipelined MIPS processor, 428
    single-cycle processor, 388–389
Product-of-sums (POS) form, 60
Program counter (PC), 310, 333, 373, *379*
Programmable logic arrays (PLAs), 67, 272–274, 588–589
    transistor-level implementation, 280
Programmable logic devices (PLDs), 588
Programmable read only memories (PROMs), 268, *270*, 584–588
Programming
    arrays. *See* Arrays
    branching. *See* Branching
    conditional statements, 316–317
    constants. *See* Constants, Immediates
    function calls. *See* Functions
    in C. *See* C programming
    in MIPS, 310–333
    instructions, 619–622
    logical instructions, 311–312
    loops. *See* Loops
    multiplication and division, 314
    shift instructions, 312–313, *312*
PROMs. *See* Programmable read only memories
Propagate signal, 241
Propagation delay, 88–92. *See also* Critical path
Pseudo-direct addressing, 334–335
Pseudo instructions, 342–343
Pseudo-nMOS logic, 33–34, *33*
    NOR gate, *33*
    ROMs and PLAs, 279–280
Pulse-Width Modulation (PWM), 536–537
    analog output with, 537
    duty cycle, 536
    signal, *536*
PWM. *See* Pulse-Width Modulation

Q

Quiescent supply current, 34

R

Race conditions, 119–120, *120*
`rand`, 662–663
Random access memory (RAM), 265–267, *271*
Read after write (RAW) hazards, 415, 451. *See also* Hazards
Read only memory (ROM), 265, *268–269*, 268–270
    transistor-level implementation, 279–280
ReadData, 378
Read/write head, 496
Receiver gate, 22
Recursive function calls, 330–332
Reduced instruction set computer (RISC), 298
Reduction operators, 180–181
RegDst, 381, 384, 397
Register file (RF)
    HDL for, 435
    in pipelined MIPS processor (write on falling edge), 412
    MIPS register descriptions, 299–300
    schematic, 267–268
    use in MIPS processor, 373
Register renaming, 452–454
Register set, 299–300. *See also* Register file
Register-only addressing, 333
Registers. *See* Flip-flops, MIPS registers, and x86 registers
Regularity, 6
RegWrite, 378, 384, 397, 413, *414*
Replacement policies, 504
Reserved segment, 337
Resettable flip-flops, 116
Resettable registers, 194–196
Resolution time, 151–152
    derivation of, 154–157
    See also Metastability
RF. *See* Register file
Ring oscillator, 119, *119*
Ripple-carry adder, *240*, 240–241, 243
Rising edge, 88
ROM. *See* Read only memory
Rotations per minute (RPM), 549
Rotators, 250–252
Rounding modes, 258
RPM. *See* Rotations per minute
RS-232, 523–524
R-type instructions, 305–306

S

Sampling, 141
Sampling rate, 531
Sampling time, *532*
SATA. *See* Serial ATA
`sb`, store byte. *See* Stores
Scalar processor, *447*
Scan chains, 261–263
`scanf`, 660
Scannable flip-flop, 262–263
Schematics, rules of drawing, 31, 67
SCK. *See* Serial Clock
SDI. *See* Serial Data In
SDO. *See* Serial Data Out
SDRAM. *See* Synchronous dynamic random access memory
Segment descriptor, 353
Segmentation, 354
Selected signal assignment statements, 182

Semiconductors, 27
  industry, sales, 3
Sequencing overhead, 143–144, 149, 160, 428
Sequential building blocks. *See* Sequential logic
Sequential logic, 109–161, 260–263
  counters, 260
  finite state machines. *See* Finite state machine
  flip-flops, 114–118. *Also see* Registers
  latches, 111–113
    D, 113
    SR, 111–113
  registers. *See* Registers
  shift registers, 261–263
  timing of. *See* Timing Analysis
Serial ATA (SATA), *562*
Serial Clock (SCK), 516
Serial communication, with PC, *525–527*
Serial Data In (SDI), *516*
Serial Data Out (SDO), *516*
Serial I/O, *515–527*
  SPI. *See* Serial peripheral interface
  UART. *See* Universal Asynchronous Receiver Transmitter
Serial Peripheral Interface (SPI), 515–521
  connection between PIC32 and FPGA, *519*
  ports
    Serial Clock (SCK), 516
    Serial Data In (SDI), 516
    Serial Data Out (SDO), 516
  register fields in, *517*
  slave circuitry and timing, *520*
  waveforms, *516*
Servo motor, *549*, *552–554*
Set bits, 483
set if less than immediate (`slti`), 345
set if less than immediate unsigned (`sltiu`), 345
set if less than (`slt`)
  circuit, 250
  in MIPS assembly, 319–320
set if less than unsigned (`sltu`), 345
Setup time constraint, 142, 145–147
  with clock skew, 148–150
Seven-segment display decoder, 79–82
  HDL for, 201–202
  with don’t cares, 82–83
SFRs. *See* Special function registers
`sh`, store half. *See* Stores
Shaft encoder, *552*, *552*
Shift instructions, 312–313, *312*
Shift registers, 261–263
Shifters, 250–252
Short path, 89–92
Sign bit, *16*
Sign extension, 18, 308
  HDL for, 436
Signed and unsigned instructions, 344–345
Signed binary numbers, 15–19
Signed multiplier, 217
Sign/magnitude numbers, 15–16, 255
Silicon dioxide (SiO2), 28
Silicon lattice, 27
SIMD. *See* Single instruction multiple data
Simple programmable logic devices (SPLDs), 274
Simulation waveforms, *176*
  with delays, *189*
Single-cycle MIPS processor, 376–389
  control, 382–385
  datapath, 376–382
  example operation, 384–385
  HDL of, 429–440
  performance, 389
Single instruction multiple data (SIMD), *447*, 454, 463
Single-precision formats, 257–258. *See also* Floating-point numbers
Skew. *See* Clock skew
Slash notation, *56*
Slave latch, 114. *See also* D flip-flop
`sll`, 312
`sllv`, 313
SLT. *See* set if less than
`slt`, set if less than, 319–320
`slti`, 345
`sltiu`, 345
`sltu`, 345
Small-scale integration (SSI) chips, 584
Solid state drive (SSD), 478–479. *See also* Flash memory and Hard drive
SOP. *See* Sum-of-products form
Spatial locality, 476, 488–490
Spatial parallelism, 157–158
Special function registers (SFRs), *509*
SPECINT2000, 406
SPI. *See* Serial Peripheral Interface
Spinstepper function, *557*
SPIxCON, *516*
Squashing, 452
SR latches, 111–113, *112*
SRAM. *See* Static random access memory
`srand`, 662–663
`srl`, 312
`srlv`, 313
SSI chips. *See* Small-scale integration
Stack, 327–333. *See also* Function calls
  during recursive function call, *331*
  preserved registers, 329–330
  stack frame, 328, *332*
  stack pointer (`$sp`), 327
  storing additional arguments on, 332–333
  storing local variables on, 332–333
Stalls, 418–421. *See also* Hazards
Standard libraries, 657–665
  math, 664–665
  stdio, 657–662
    file manipulation, 660–662
    `printf`, 657–659
    `scanf`, 660
  stdlib, 662–664
    `exit`, 663
    format conversion (`atoi`, `atol`, `atof`), 663–664
    `rand`, `srand`, 662–663
  string, *665*
State encodings, FSM, 129–131, *134*. *See also* Binary encoding, One-cold encoding, One-hot encoding
State machine circuit. *See* Finite state machines
State variables, 109
Static branch prediction, 446
Static discipline, 24–26
Static power, 34
Static random access memory (SRAM), 266, *267*
Status flags, 350
stdio.h, C library, 657–662. *See also* Standard libraries
stdlib.h, C library, 662–664. *See also* Standard libraries
Stepper motors, *549*, *554–556*
  bipolar stepper motor, *554–555*
    half-step drive, *554*
    two-phase-on drive, *554*
    wave drive, *554*
Stored program, 309–310

- Stores
  - store byte (`sb` or `sbu`), 302–304, 323–324
  - store half (`sh` or `shu`), 345
  - store word (`sw`), 302–304
- string.h, C library, 665
- Strings, 324, 650–651. *See also* Characters (`char`)
- Structural modeling, 173–174, 190–193
- Structures (`struct`), 651–653
- `sub`, 297
- Substrate, 28–29
- Subtraction, 17, 246, 297
  - signed and unsigned instructions, 344–345
- Subtractor, 246–247
- `subu`, 345
- Sum-of-products (SOP) form, 58–60
- Superscalar processor, 447–449
- Supply voltage, 22. *See also* V_{DD}
- `sw`, store word, 302–304. *See also* Stores
- Swap space, 504
- Switch/case statements
  - in C, 639–640
  - in HDL. *See* Case statement
  - in MIPS assembly, 317
- Symbol table, *339*
- Symmetric multiprocessing (SMP). *See* Homogeneous multiprocessors
- Synchronizers, 152–154, *152–153*
- Synchronous circuits, 122–123
- Synchronous dynamic random access memory (SDRAM), 267
  - DDR, 267
- Synchronous logic, design, 119–123
- Synchronous resettable flip-flops, 116
- Synchronous sequential circuits, 120–123, *122*. *See also* Finite state machines
  - timing specification. *See* Timing analysis
- Synergistic processor elements (SPEs), 457
- Synergistic Processor Unit (SPU) ISA, *458*
- SystemVerilog, 173–225. *See also* Hardware description languages
  - accessing parts of busses, 188, 192
  - bad synchronizer with blocking assignments, 209
  - bit swizzling, 188
  - blocking and nonblocking assignment, 199–200, 205–208
  - case statements, 201–202, 205
  - combinational logic using, 177–193, 198–208, 217–220
  - comments, 180
  - conditional assignment, 181–182
  - data types, 213–217
  - decoders, 202–203, 219
  - delays (in simulation), 189
  - divide-by-3 FSM, 210–211
  - finite state machines (FSMs), 209–213
    - Mealy FSM, 213
    - Moore FSM, 210, 212
  - full adder, 184
    - using always/process, 200
    - using nonblocking assignments, 208
  - history of, 175
  - if statements, 202–205
  - internal signals, 182–184
  - inverters, 178, 199
  - latches, 198
  - logic gates, 177–179
  - multiplexers, 181–183, 190–193, 218–219
  - multiplier, 217
  - numbers, 185–186
  - operators, *185*
  - parameterized modules, 217–220
    - N:2^N decoder, 219
    - N-bit multiplexers, 218–219
    - N-input AND gate, 220
  - priority circuit, 204
    - using don't cares, 205
  - reduction operators, 180–181
  - registers, 193–197
    - enabled, 196
    - resettable, 194–196
  - sequential logic using, 193–198, 209–213
  - seven-segment display decoder, 201
  - simulation and synthesis, 175–177
  - structural models, 190–193
  - synchronizer, 197
  - testbench, 220–224, 437–438
    - self-checking, 222
    - simple, 221
    - with test vector file, 223–224
  - tristate buffer, 187
  - truth tables with undefined and floating inputs, 187, *188*
  - z's and x's, 186–188, 205

T

- Tag, 483
- Taking the two's complement, 16–17
- Temporal locality, 476, 481–482, 485, 490
- Temporal parallelism, 158–159
- Temporary registers, 299, 329–330
- Ternary operators, 181, 635
- Testbenches, HDLs, 220–224
  - for MIPS processor, 437–438
  - simple, 220–221
  - self-checking, 221–222
  - with testvectors, 222–224
- Text segment, 336, 340
- Thin Quad Flat Pack (TQFP), 510
- Thin small outline package (TSOP), 599
- Thread level parallelism (TLP), 455
- Threshold voltage, 29
- Throughput, 157–160, 374–375, 409–411, 455
- Timers, 527–529
  - delay generation using, 528–529
- Timing
  - of combinational logic, 88–95
    - delay. *See* Propagation delay, Contamination delay
    - glitches. *See* Glitches
  - of sequential logic, 141–157
    - analysis. *See* Timing analysis
    - clock skew. *See* Clock skew
    - dynamic discipline, 141–142
    - metastability. *See* Metastability
    - resolution time. *See* Resolution time
    - system timing. *See* Timing analysis
- Timing analysis, 141–151
  - calculating cycle time. *See* Setup time constraint
  - hold time constraint. *See* Hold time constraint
  - max-delay constraint. *See* Setup time constraint
  - min-delay constraint. *See* Hold time constraint
  - multi-cycle processor, 407–408
  - pipelined processor, 428
  - setup time constraint. *See* Setup time constraint
  - single-cycle processor, 388–389
  - with clock skew. *See* clock skew

- TLB. *See* Translation lookaside buffer
- Trace cache, 463
- Transistors, 26–34
  - bipolar, 26
  - CMOS, 26–33
  - gates made from, 31–34
  - latches and flip-flops, 116–117
  - MOSFETs, 26
  - nMOS, 28–34, *29–33*
  - pMOS, 28–34, *29–33*
    - pseudo-nMOS, 33–34
    - ROMs and PLAs, 279–280
    - transmission gate, 33
- Transistor-Transistor Logic (TTL), 25–26, 597–598
- Translating and starting a program, 337–342, *338*
- Translation lookaside buffer (TLB), 502–503
- Transmission Control Protocol and Internet Protocol (TCP/IP), 561
- Transmission gates, 33
- Transmission lines, 602–615
  - characteristic impedance (Z_0), 612–613
    - derivation of, 612–613
  - matched termination, 604–606
  - mismatched termination, 607–610
  - open termination, 606–607
  - reflection coefficient (k_r), 613–614
    - derivation of, 613–614
  - series and parallel terminations, 610–612
  - short termination, 607
  - when to use, 610
- Transparent latch. *See* D latch
- Traps, 343
- Tristate buffer, 74–75, *187*
  - HDL for, 186–187
  - multiplexer built using, 84–85, 91–93
- Truth tables, 20
  - ALU decoder, 383, *384*
  - multiplexer, 83
  - seven-segment display decoder, 79
  - SR latch, 111, *112*
  - with don't cares, 69, 81–83, 205
  - with undefined and floating inputs, *187–188*
- TSOP. *See* Thin small outline package
- TTL. *See* Transistor-Transistor Logic
- Two-bit dynamic branch predictor, 447
- Two-cycle latency of `lw`, 418
- Two-level logic, 69
- Two's complement numbers, 16–18
- `typedef`, 653–654

U

- UART. *See* Universal Asynchronous Receiver Transmitter
- Unconditional branches, 315–316
- Undefined instruction exception, 343–344, 440–443
- Unicode, 322
- Unit under test (UUT), 220
- Unity gain points, 24
- Universal Asynchronous Receiver Transmitter (UART), 521–527
  - hardware handshaking, 523
  - STA register, 524
- Universal Serial Bus (USB), *270*, 523, 559–560
  - USB 1.0, 560
  - USB 2.0, 560
  - USB 3.0, 560
- Unsigned multiplier, *217*
- Unsigned numbers, 18
- USB. *See* Universal Serial Bus
- USB links, 563–564
  - FTDI, 563
    - UM232H module, *564*
- Use bit (U), 490

V

- Valid bit (V), 484
- Vanity Fair (Carroll), *76*
- Variables in C, 629–633
  - global and local, 631–632
  - initializing, 633
  - primitive data types, 630–631
- Variable-shift instruction, 313
- V_{CC}, *23*. *See also* Supply voltage, V_{DD}
- V_{DD}, 22, *23*. *See also* Supply voltage
- Vector processor, *447*
- Verilog. *See* SystemVerilog
- Very High Speed Integrated Circuits (VHSIC), 175. *See also* VHDL
- VGA. *See* VGA monitor
- VGA (Video Graphics Array) monitor, 541–547
  - connector pinout, *543*
  - driver for, *544–547*
- VHDL. *See* VHSIC Hardware Description Language
- VHSIC. *See* Very High Speed Integrated Circuits
- VHSIC Hardware Description Language (VHDL), 173–175
  - accessing parts of busses, 188, 192
  - bad synchronizer with blocking assignments, 209
  - bit swizzling, 188
  - blocking and nonblocking assignment, 199–200, 205–208
  - case statements, 201–202, 205
  - combinational logic using, 177–193, 198–208, 217–220
  - comments, 180
  - conditional assignment, 181–182
  - data types, 213–217
  - decoders, 202–203, 219
  - delays (in simulation), 189
  - divide-by-3 FSM, 210–211
  - finite state machines (FSMs), 209–213
    - Mealy FSM, 213
    - Moore FSM, 210, 212
  - full adder, 184
    - using always/process, 200
    - using nonblocking assignments, 208
  - history of, 175
  - if statements, 202
  - internal signals, 182–184
  - inverters, 178, 199
  - latches, 198
  - logic gates, 177–179
  - multiplexer, 181–183, 190–193, 218–219
  - multiplier, 217
  - numbers, 185–186
  - operators, *185*
  - parameterized modules, 217–220
    - $N{:}2^N$ decoder, 219
    - N-bit multiplexers, 218, *219*
    - N-input AND gate, 220, *220*
  - priority circuit, 204
    - reduction operators, 180–181
    - using don't cares, 205

VHSIC Hardware Description Language (VHDL) (*Continued*)
  reduction operators, 180–181
  registers, 193–197
    enabled, 196
    resettable, 194–196
  sequential logic using, 193–198, 209–213
  seven-segment display decoder, 201
  simulation and synthesis, 175–177
  structural models, 190–193
  synchronizer, 197
  testbench, 220–224, 437–438
    self-checking, 222
    simple, 221
    with test vector file, 223–224
  tristate buffer, 187
  truth tables with undefined and floating inputs, 187, *188*
  z's and x's, 186–188, 205
Video Graphics Array (VGA). *See* VGA monitor
Virtual address, 497
  space, 503
Virtual memory, 478, 496–506
  address translation, 497–500
  cache terms comparison, 497
  memory protection, 503
  multilevel page tables, 504–506
  page fault, 497
  page number, 498
  page offset, 498
  page table, 500–501
  pages, 497
  replacement policies, 504
  translation lookaside buffer (TLB), 502–503
  write policy, 494–495
  x86, 567. *See also* x86
Virtual page number (VPN), 499
Virtual pages, 497
V_{SS}, *23*

W

Wafers, 28
Wait states, *564*
Wall, Larry, *20*
WAR. *See* Write after read
WAW. *See* Write after write
Weak pull-up, 33
Weird number, 18
While loops, 318–319, 641
White space, 180
Whitmore, Georgiana, *7*
Wi-Fi, 561
Wire, 67
Wireless communication, Bluetooth, 547–548
Word-addressable memory, *301*, 302
Wordline, 264
Write after read (WAR) hazard, 451–453. *See also* Hazards
Write after write (WAW) hazard, 451
Write policy, 494–495
  write-back, 494–495
  write-through, 494–495

X

X. *See* Contention, Don't care
x86
  architecture, 347–355
    branch conditions, *352*
    instruction encoding, 352–354, *353*
    instructions, 350–352, *351*
    memory addressing modes, *349*
    operands, 348–350
    registers, 348
    status flags, 350
    vs. MIPS, *348*
  cache systems, 564–567
  memory system, evolution of, *565*
  microarchitecture, 458–465
    evolution of, 458–459
  programmed I/O, 567–568
  registers, 348
  virtual memory, 567
    protected mode, 567
    real mode, 567
Xilinx FPGA, 274–276
XNOR gate, 21–22
XOR gate, 21
`xor`, 311
`xori`, 311–312

Z

Z. *See* Floating
Zero extension, 250, 308, 311–312, 345

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank